COMPUTATIONWORLD 2020

Escaping Groundhog Day

Embedding Business Objects in an Evolving Landscape of Cross-Cutting Concern Utilities

HERWIG MANNAERT

OCTOBER, 2020

Universiteit Antwerpen

ESCAPING GROUNDHOG DAY

Overview

- Groundhog Day
- Foundations of Evolvable Software
- Toward Scalable Metaprogramming
- A Glimpse Beyond Software
- Conclusion

ESCAPING GROUNDHOG DAY

Overview

- Groundhog Day
    - Distributed Business Services
- Foundations of Evolvable Software
- Toward Scalable Metaprogramming
- A Glimpse Beyond Software
- Conclusion

Groundhog Day

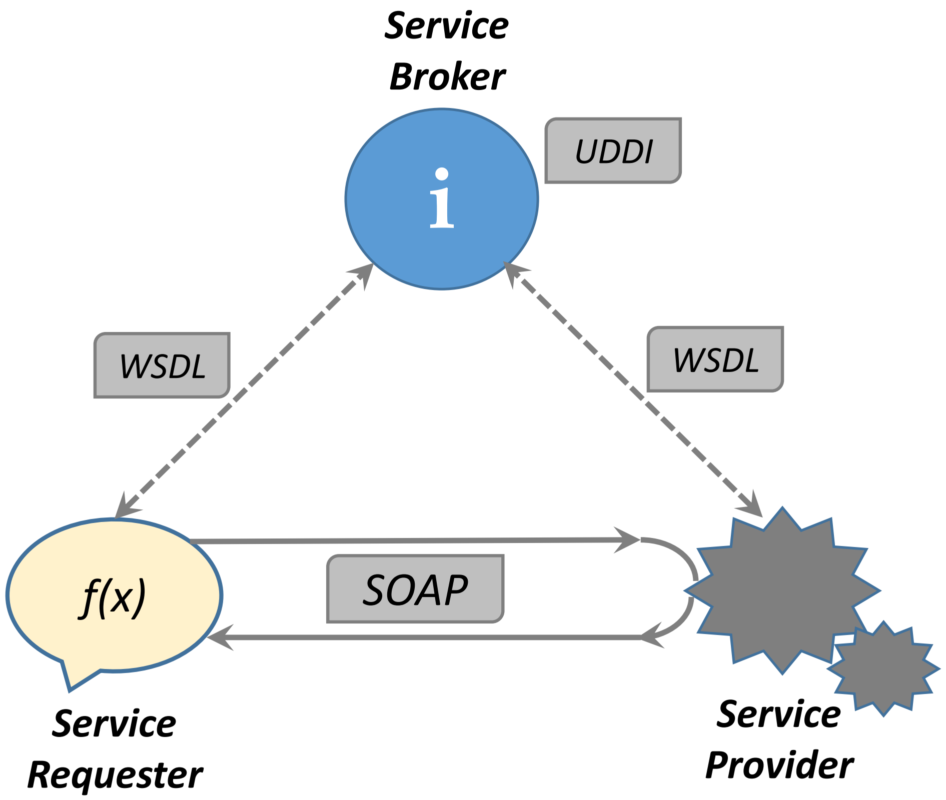
- Popular American tradition
  - Mainly in Pennsylvania *(Punxsutawney)*
  - Based on groundhog emerging from its burrow
- Fantasy comedy from 1993
  - Man becomes trapped in a time loop
  - Forced to relive February 2 over and over again
- Became part of the English lexicon
  - Monotonous, unpleasant, and repetitive situation
  - *Series of unwelcome or tedious events appear to be recurring in exactly the same way*

The Quest for Distributed Plug & Play

- Monolithic applications dominated 1960’s and 1970’s
- Distributed architectures & standards emerged from 1980’s:
  - DCE/RPC in 1980’s – 1990’s
  - CORBA in 1990’s – 2000’s
  - XML/RPC – Web Services in 2000’s – 2010’s
  - JSON/RPC – REST Services in 2010’s – 2020’s
  - *Service Mesh – Sidecar Proxy in 2020’s …*
- Business objects and capabilities
  - are (re-)implemented *in a repetitive way*
  - in an often *tedious and recurring way*

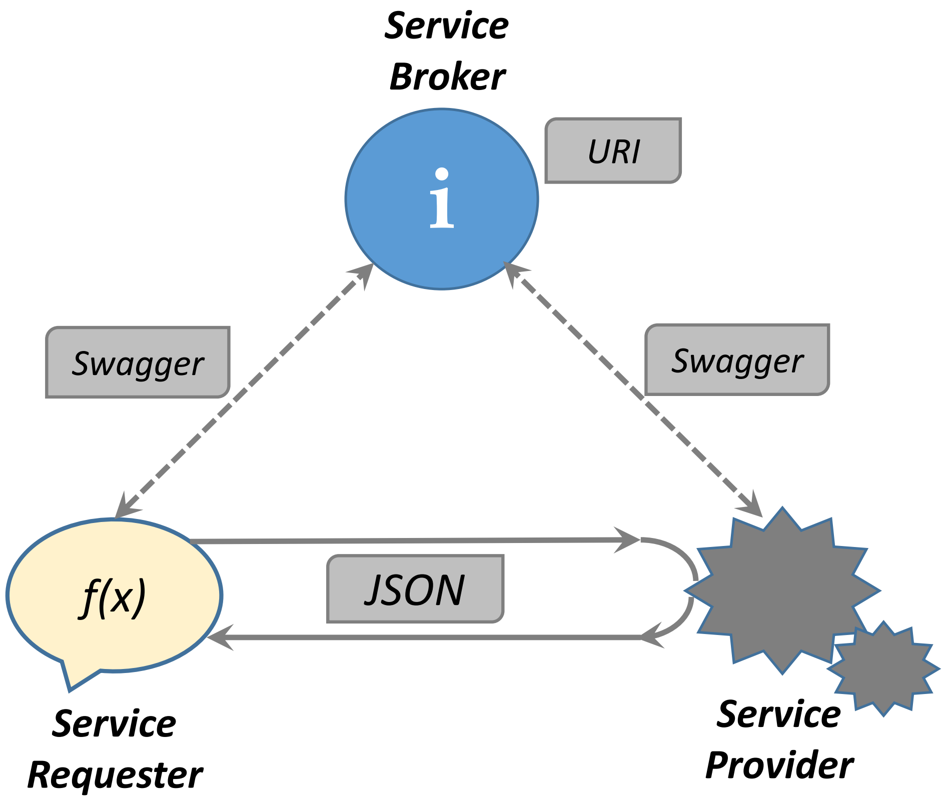
XML/RPC – Web Services

- Participants
  - Service Provider
  - Service Requester
  - Service Broker
- Protocols
  - SOAP: HTTP – XML
  - WSDL
  - UDDI



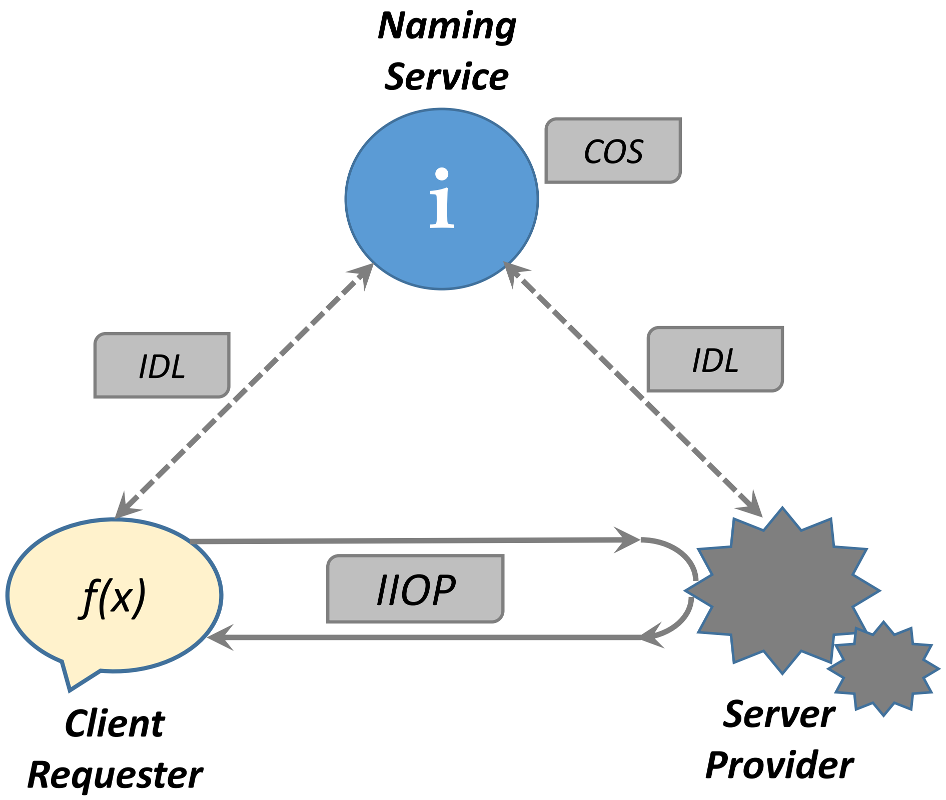
JSON/RPC – REST Services

- Participants
  - Service Provider
  - Service Requester
  - Service Broker
- Protocols
  - HTTP – JSON
  - Swagger
  - URI



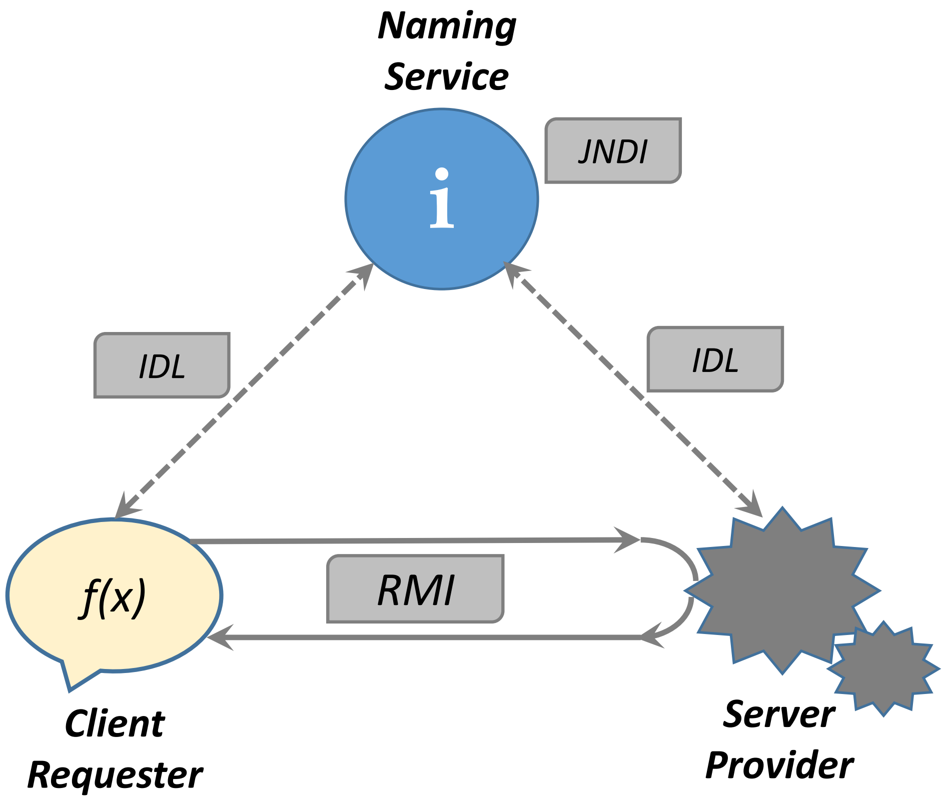
Common Object Request Broker Architecture

- Participants
  - Client Requester
  - Server Provider
  - Naming Service
- Protocols
  - IIOP
  - IDL
  - COS



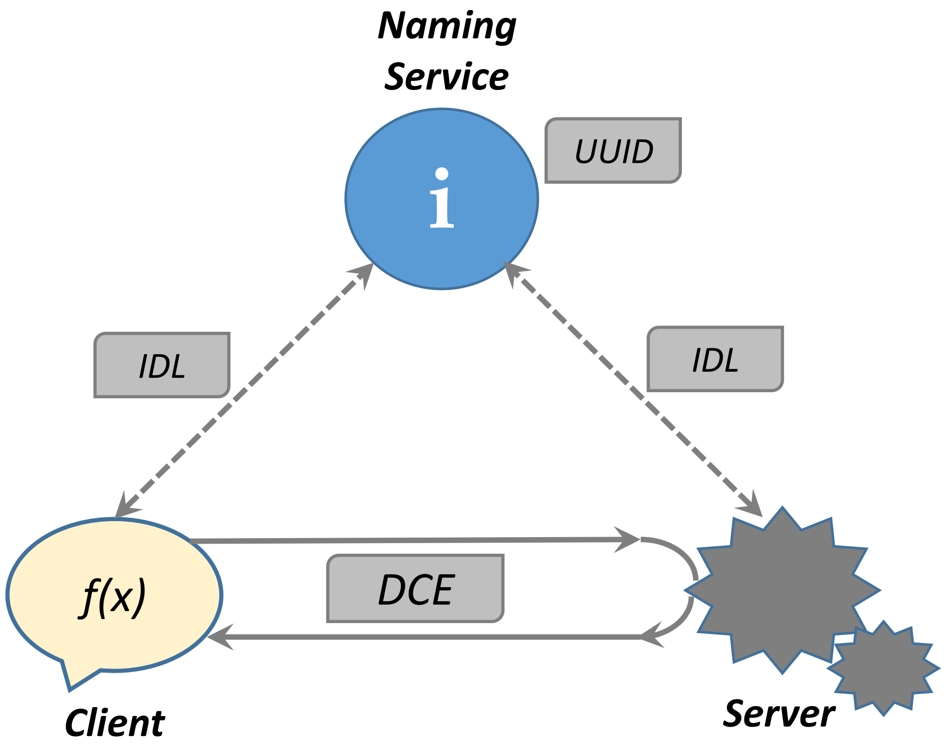
Java Remote Method Invocation

- Participants
  - Client Requester
  - Server Provider
  - Naming Service
- Protocols
  - RMI
  - Java IDL
  - JNDI



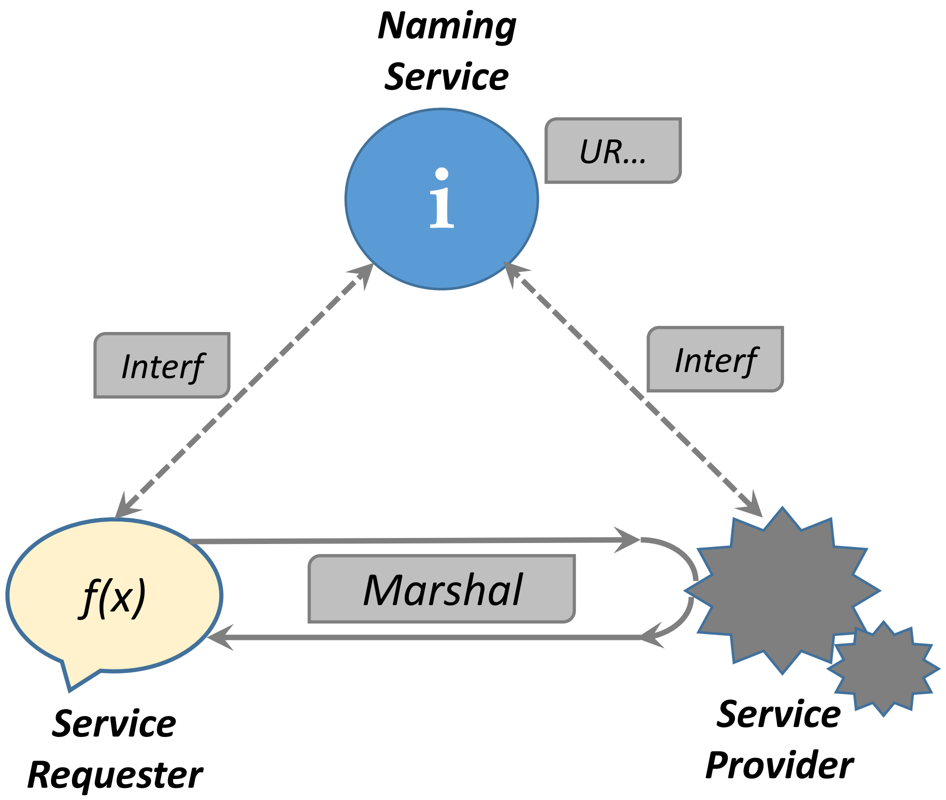
DCE / Remote Procedure Call

- Participants
  - Server
  - Client
  - Directory Service
- Protocols
  - DCE
  - IDL
  - UUID



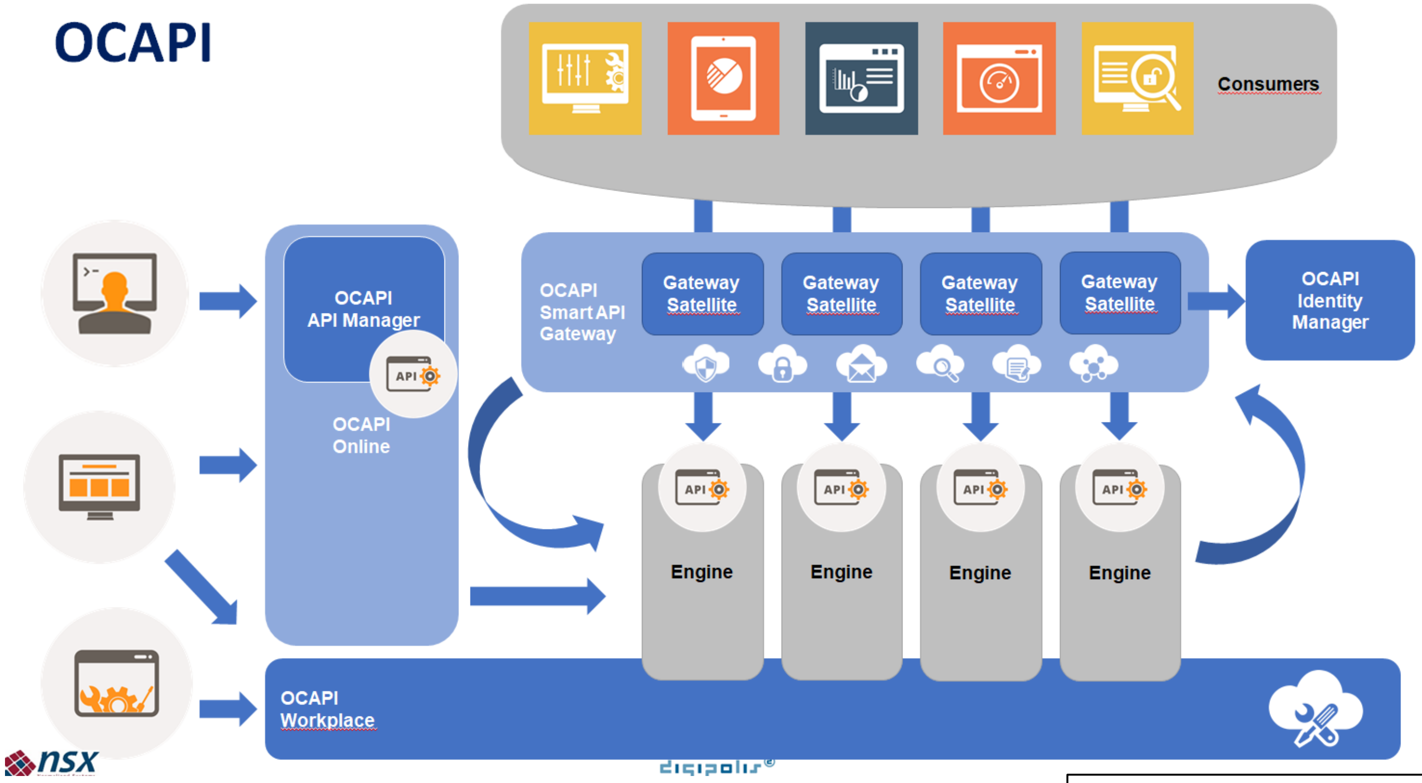
Software as a Service

- New service infrastructures emerge, based on:
  - Cloud Computing
  - Containerization
  - Serverless
  - Datastores
  - Service Mesh
  - Side-car Proxy
  - …
- But basically …



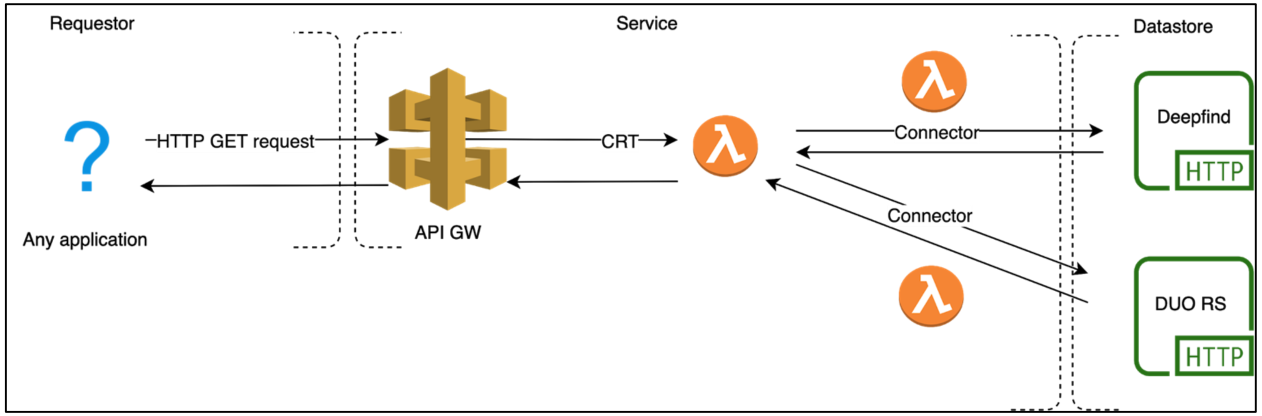
SaaS Platforms – Some Recent Cases

OCAPI



Local government

HR Company

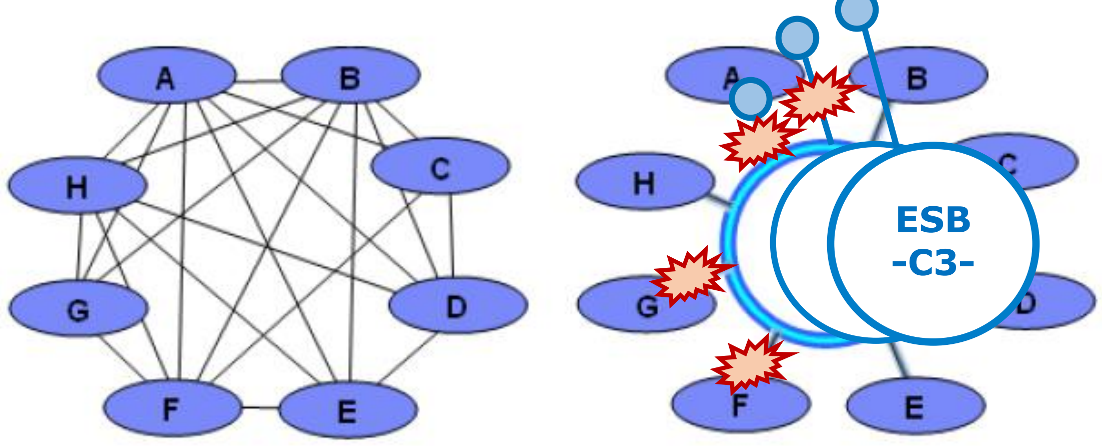


About the Tedious and Recurring Nature

- The business logic is not cleanly separated from the middleware
- Communication middleware protocol is not the only *bus dimension*:
  - Authorization
  - Access control
  - Authentication
  - Load balancing
  - Logging, archiving
  - …
- All these concerns are intertwined with business logic, causing:
  - Duplication of cross-cutting concerns
  - Duplication of business logic

i.e., the Enterprise Service Bus Fallacy …

http://nl.wikipedia.org/wiki/Enterprise_Service_Bus



Peer to Peer	Vs.	Integration Bus
N(N-1) / 2	*# links total*	N
N	*# links / node*	1
2	*# concerns / link*	1

We need multiple ESB's, i.e., for every concern

The encapsulations do not realize the decoupling

- Groundhog Day
- Foundations of Evolvable Software
  - Stability
  - Regeneration
  - Meta-Circularity
- Toward Scalable Metaprogramming
- A Glimpse Beyond Software
- Conclusion

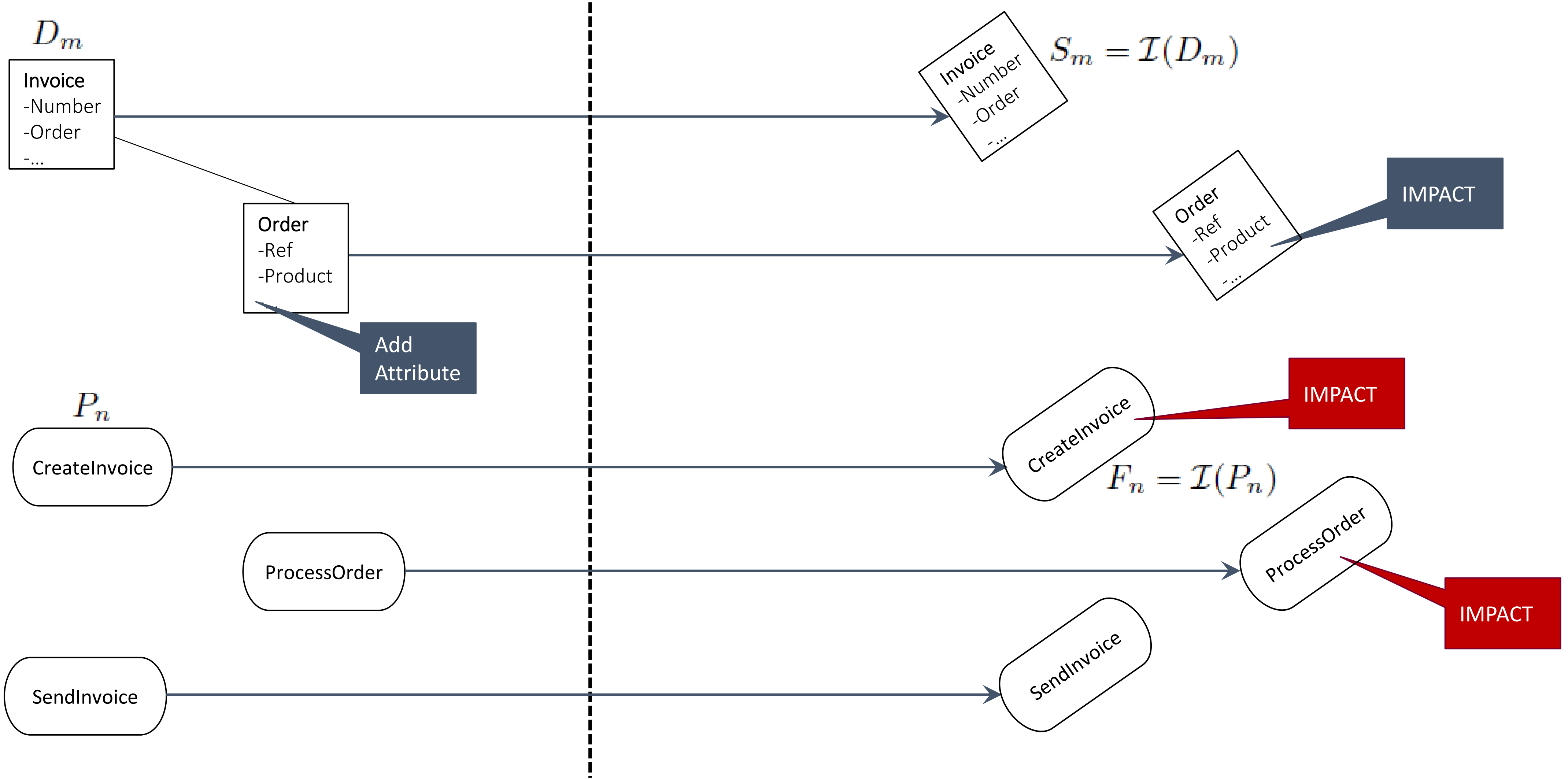
ESCAPING GROUNDHOG DAY

Overview

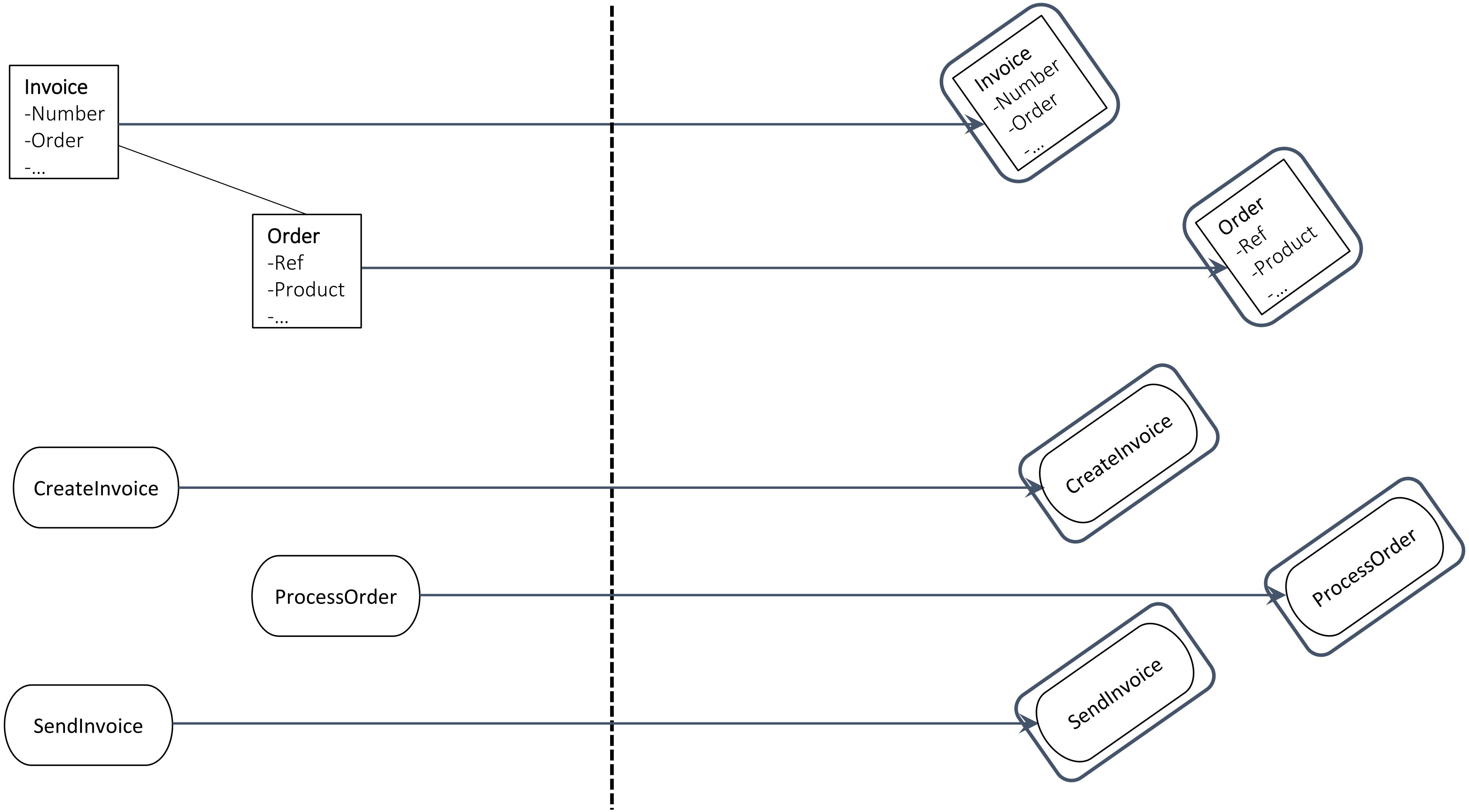
Design Theorems for Stable Software

- In order to avoid dynamic instabilities in the software design cycle, the *rippling of changes needs to be depleted or damped: a = 0*
- As these ripples create *combinations of multiple changes* for every functional change, we call these instabilities ***combinatorial effects***
- Demanding systems theoretic stability for the software transformation, leads to the derivation of ***principles*** in line with existing heuristics
- Adhering to these principles avoids dynamic instabilities, meaning that these *principles are necessary, not sufficient for systems stability*

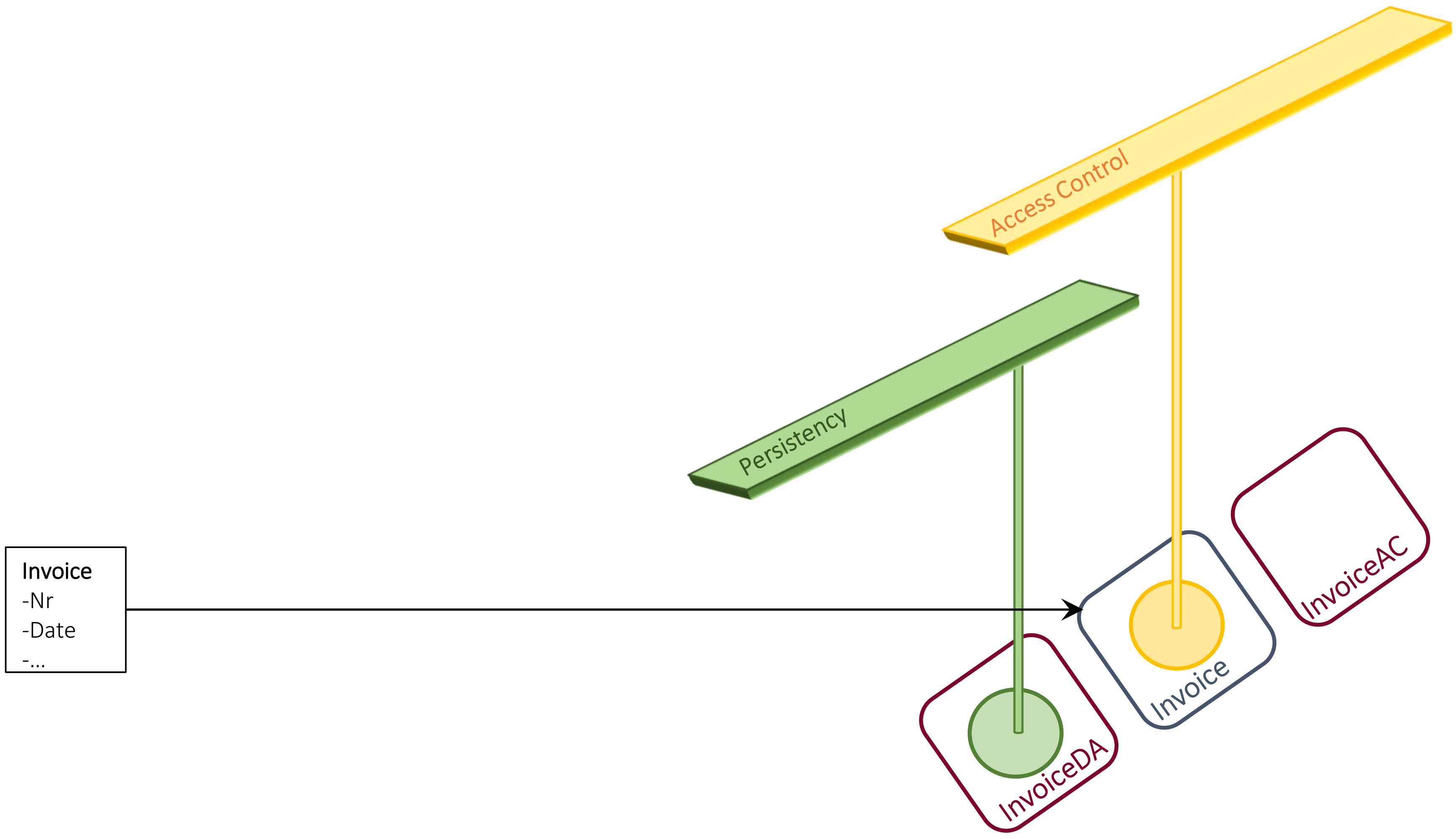
Change Ripples: A Basic Transformation



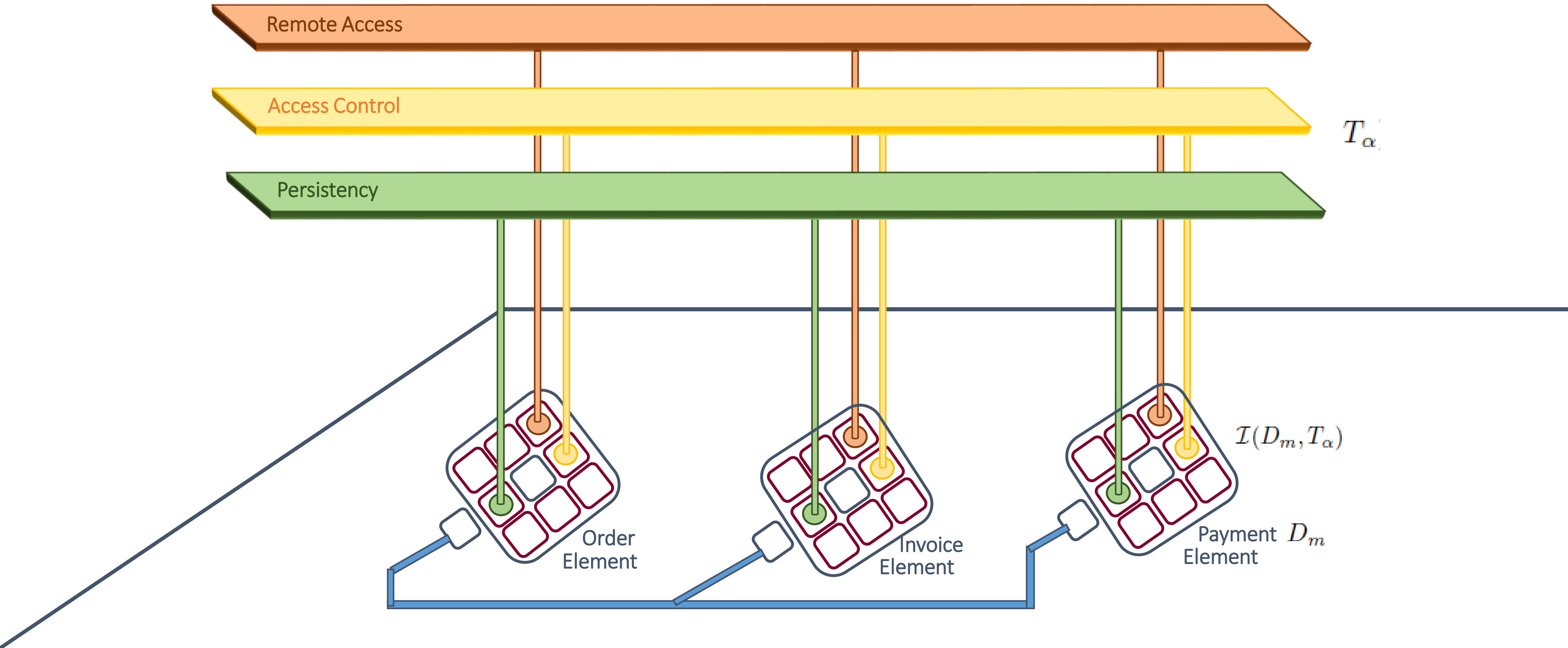
Encapsulating Basic Primitives



Separating Cross-Cutting Concerns

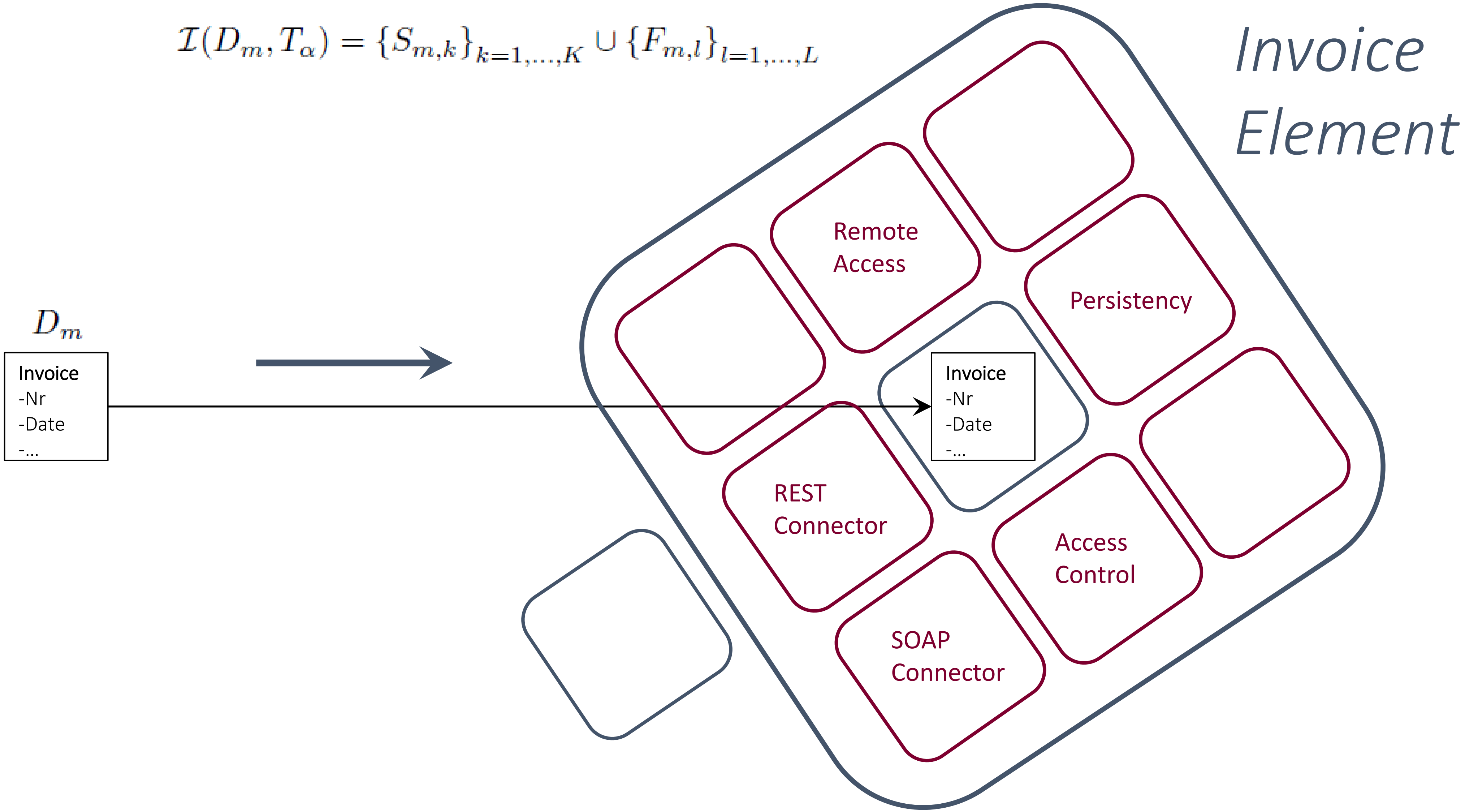


The Emergence of Elements

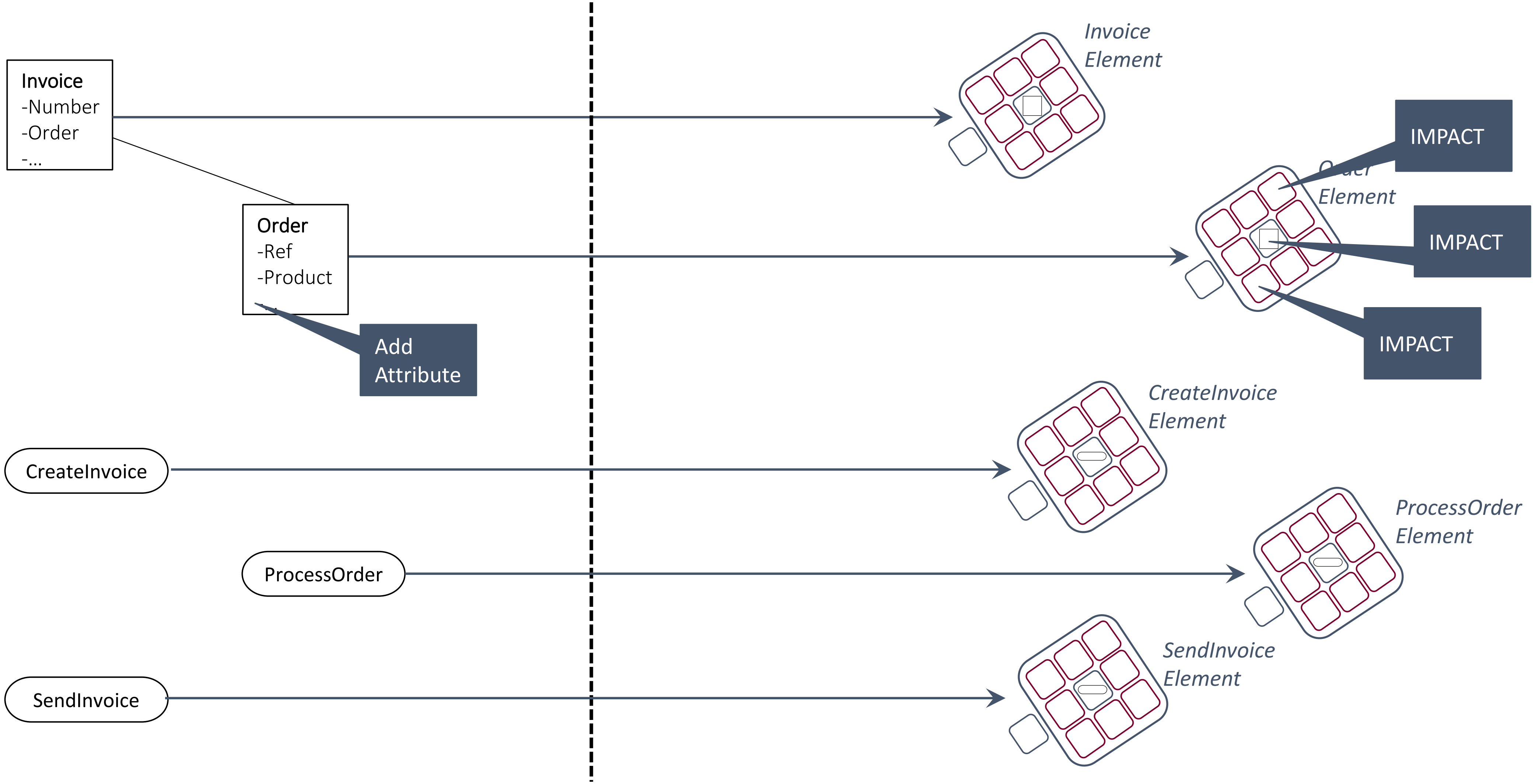


An Advanced Transformation

$$\mathcal{I}(D_m, T_\alpha) = \{S_{m,k}\}_{k=1,\ldots,K} \cup \{F_{m,l}\}_{l=1,\ldots,L}$$



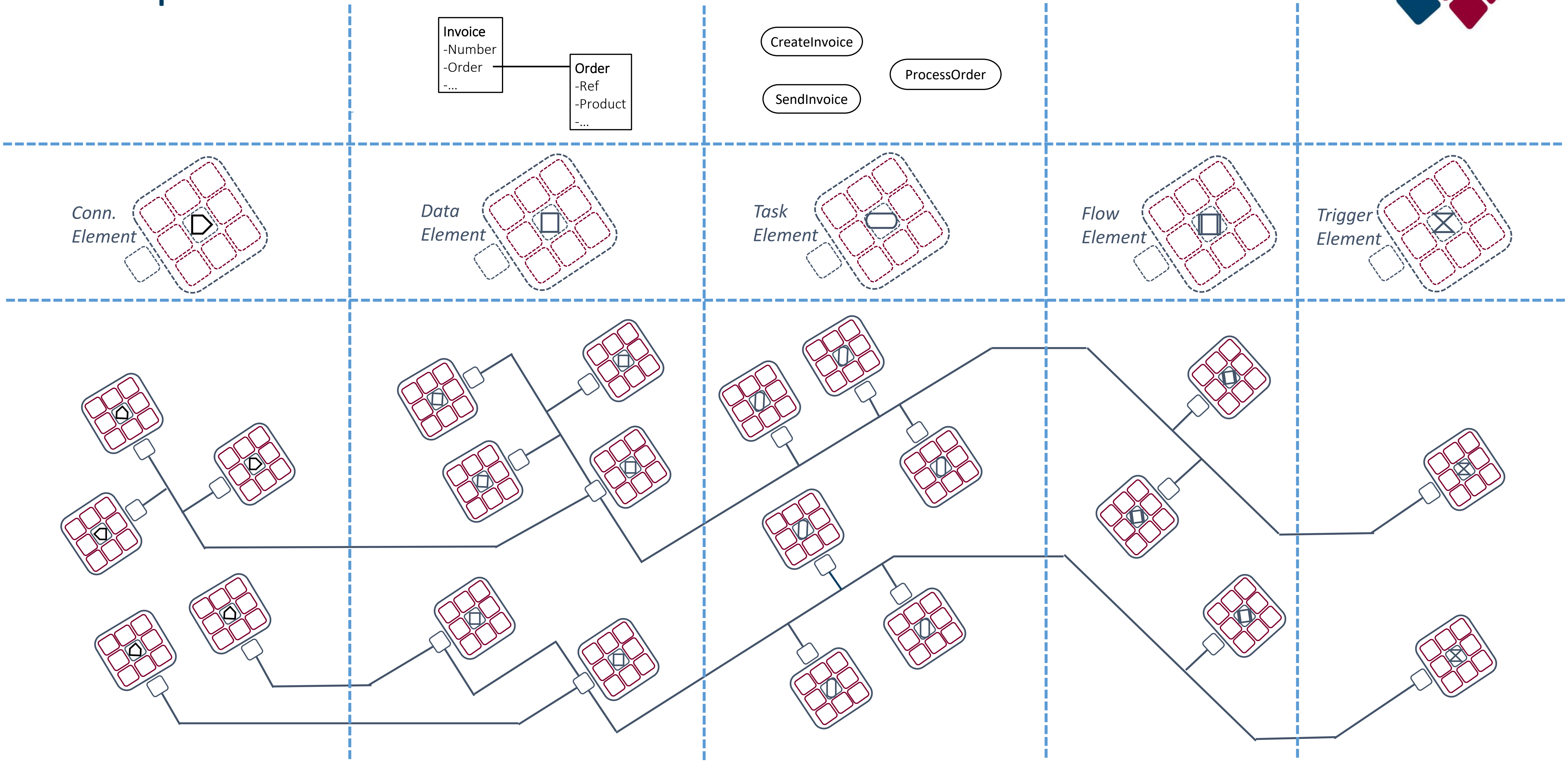
An Advanced Transformation



Normalized Systems Elements

- Element structures are needed *to interconnect with CCC solutions*
- NS defines 5 types of elements, aligned with basic software concepts:
  - *Data elements*, to represent data variables and structures
  - *Task elements*, to represent instructions and/or functions
  - *Flow elements*, to handle control flow and orchestrations
  - *Connector elements*, to allow for input/output commands
  - *Trigger elements*, to offer periodic clock-like control
- It seems obvious to use code generation techniques to create instances of these recurrent element structures
- Due to its simple and deterministic nature, we refer to this process as *expansion*, and to the generators as *expanders*

Expansion of Elements



- Groundhog Day
- Foundations of Evolvable Software
  - Stability
  - Regeneration
  - Meta-Circularity
- Toward Scalable Metaprogramming
- A Glimpse Beyond Software
- Conclusion

ESCAPING GROUNDHOG DAY

Overview

On Updating Recurring Structure

- Structure should be recurring, as variations:
  - increase complexity of codebase
  - decrease consistency in behaviour
- Recurring structure may need to vary over time:
  - new insights
  - discovery of flaws
  - changes in technologies

Structural changes may need to be applied with retroactive effect, but the efforts increase with the frequency of change.

N instances, update every K ➔ ***#updates*** = $\frac{N(N+K)}{2K}$

K=50	K=20	K=10	K=5
			5
		10	10
			15
	20	20	20
			25
		30	30
			35
	40	40	40
			45
50		50	50
			55
	60	60	60
			65
		70	70
			75
	80	80	80
			85
		90	90
			95
100	100	100	100
150	300	550	1050

N=100

K	Total
100	100
50	150
20	300
10	550
5	1050
2	2550
1	5050

Catch 22: The Only Way Out

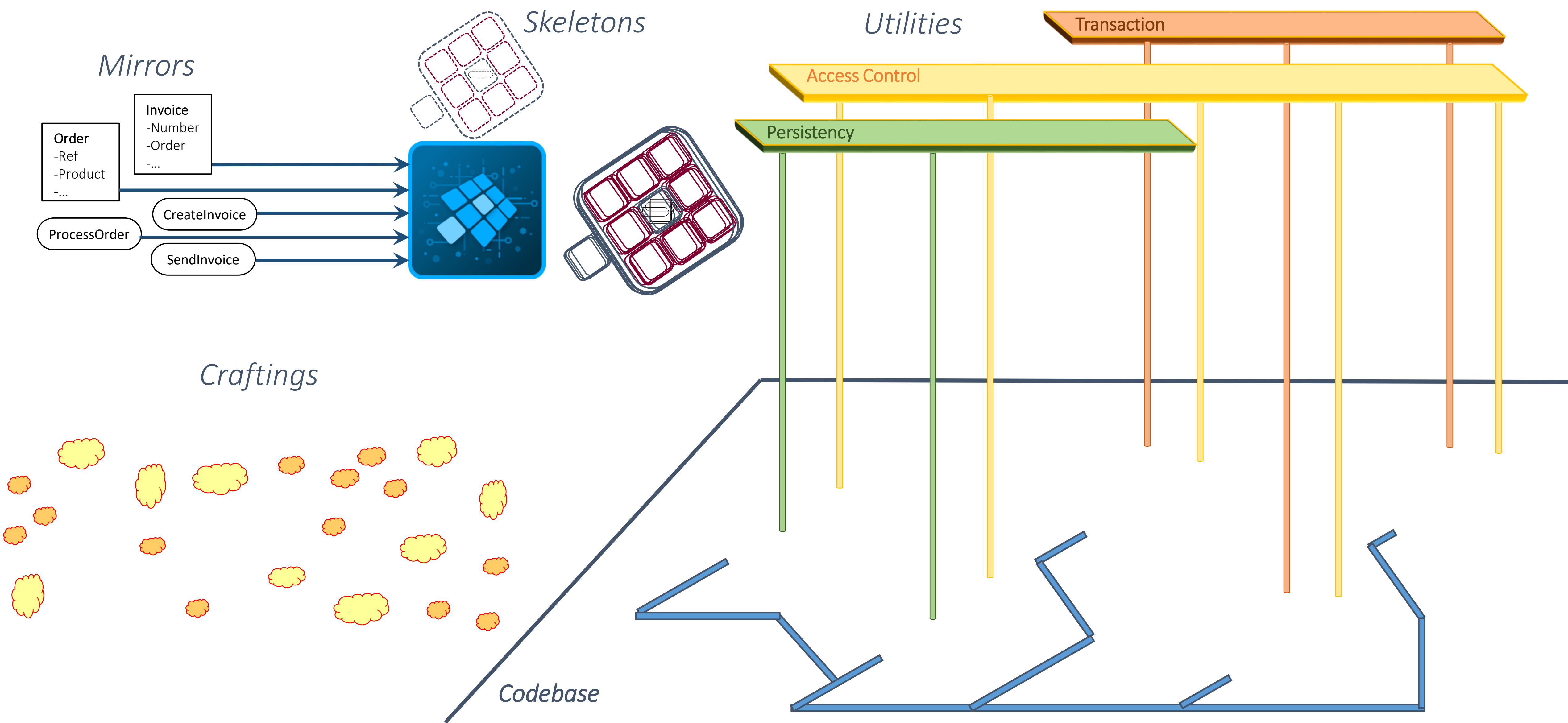
- Recurrent stable structures are required to limit complexity and to guarantee consistency
- Recurrent stable structures need to be able to adapt over time, to overcome flaws and technology changes
- Additional custom code is inevitable and needs to be maintained across updated stable structures

An automated mechanism is required,
providing both code generation or expansion,
and regeneration with harvesting and injection.

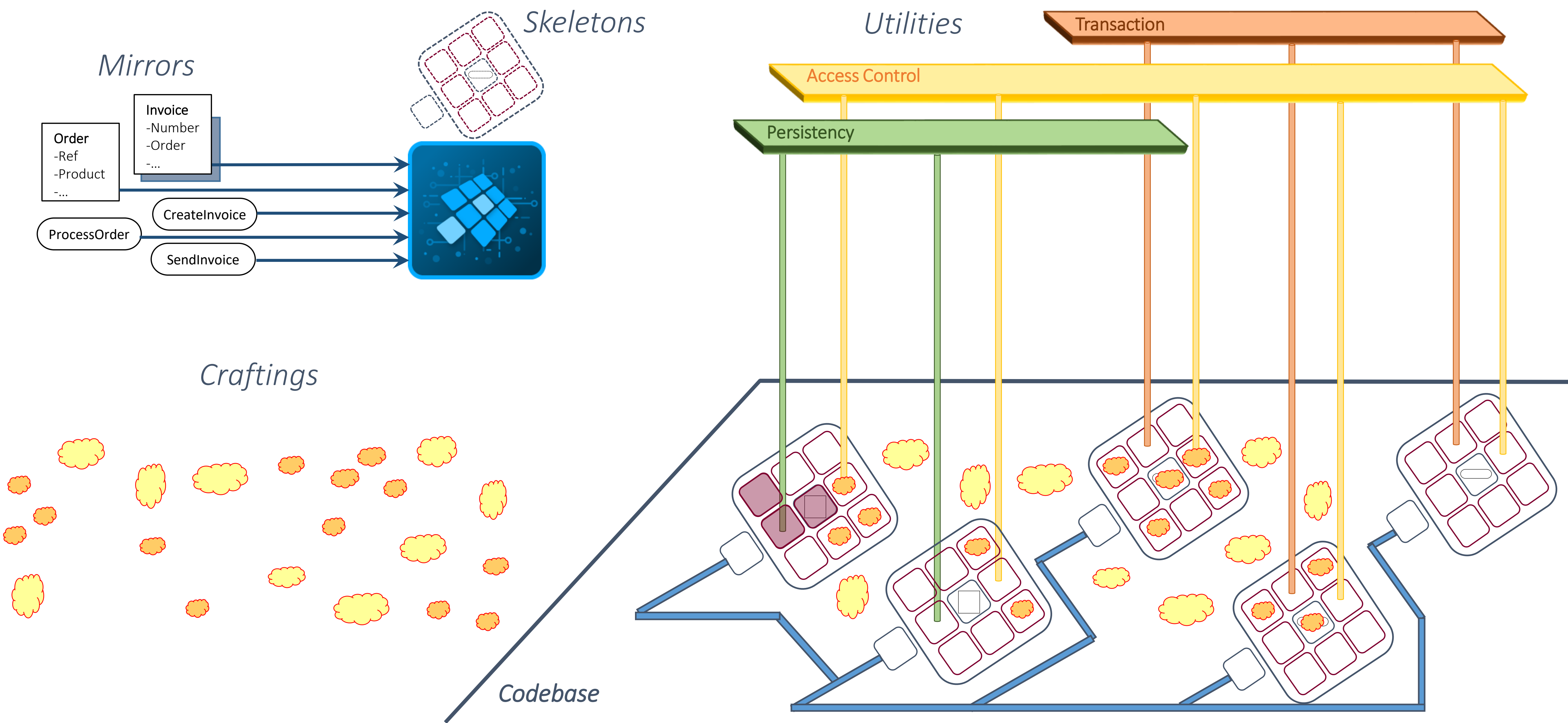
Variability Dimensions and Expansion

- We identify four dimensions of variability:
  - **M**odels or *mirrors*, new data attributes/relations, new elements
  - **E**xpanders or *skeletons*, new or improved implementations of concerns
  - **I**nfrastructure or *utilities*, new frameworks to implement various concerns
  - **C**ustom code or *craftings*, new or improved implementations of tasks, screens
- *If separated and well encapsulated*
  - Number of versions to maintain is *additive*: $\#V = \#M + \#E + \#I + \#C$
  - Number of versions available is *multiplicative*: $\#V = \#M \text{ x } \#E \text{ x } \#I \text{ x } \#C$
  - Where the same holds within any individual dimensions,
    e.g., infrastructure dimension: $\#I = \#G \text{ x } \#P \text{ x } \#B \text{ x } \#T$

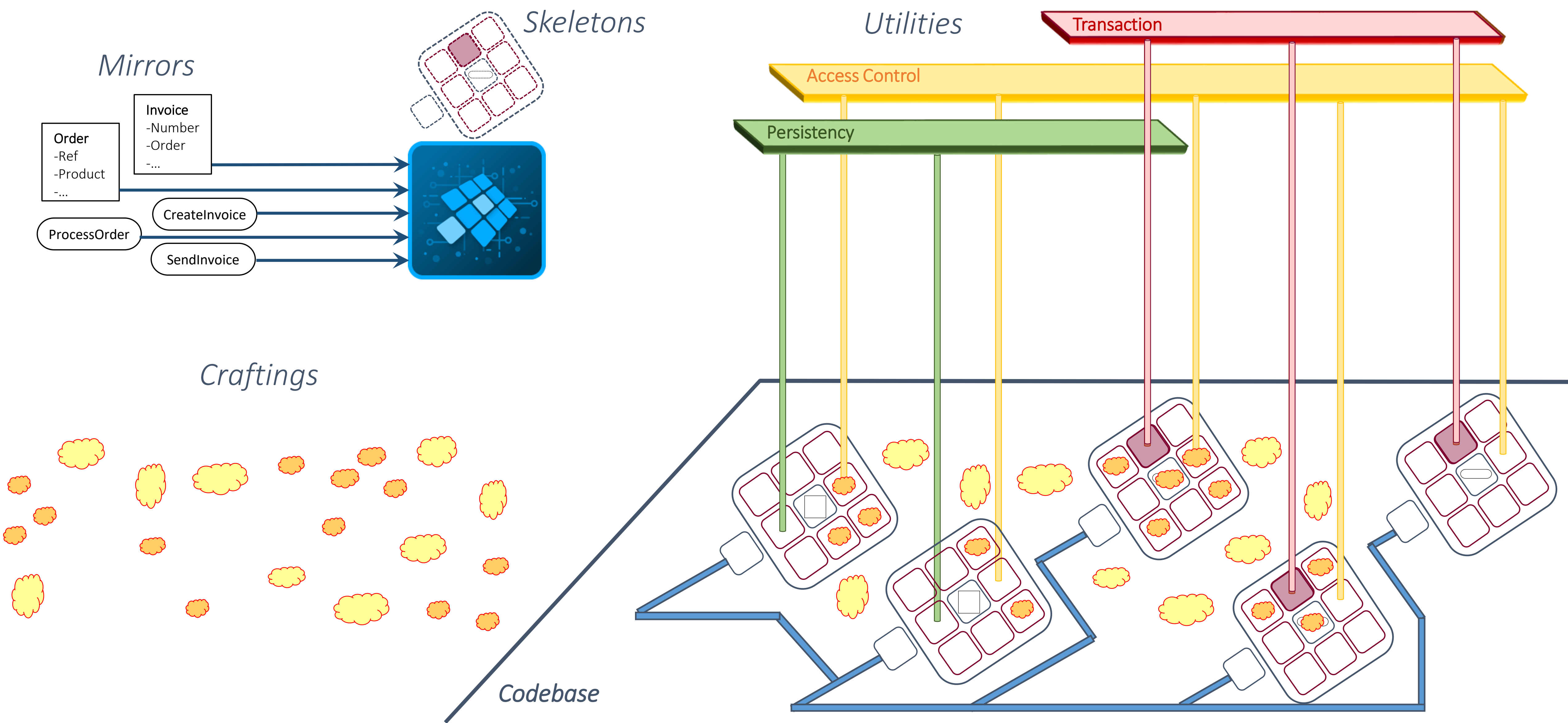
Integrating the Dimensions of Variability



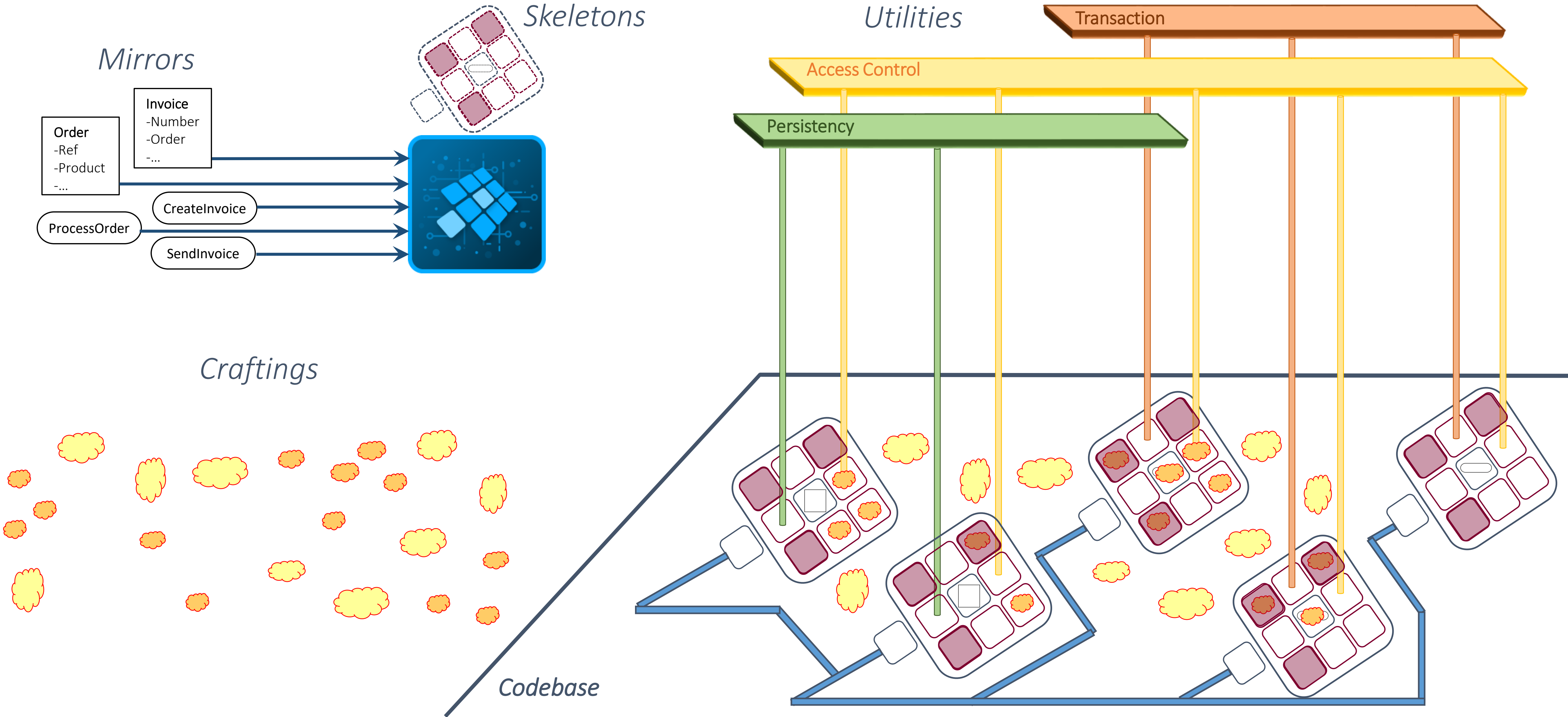
Change Dimension 1: The Mirrors



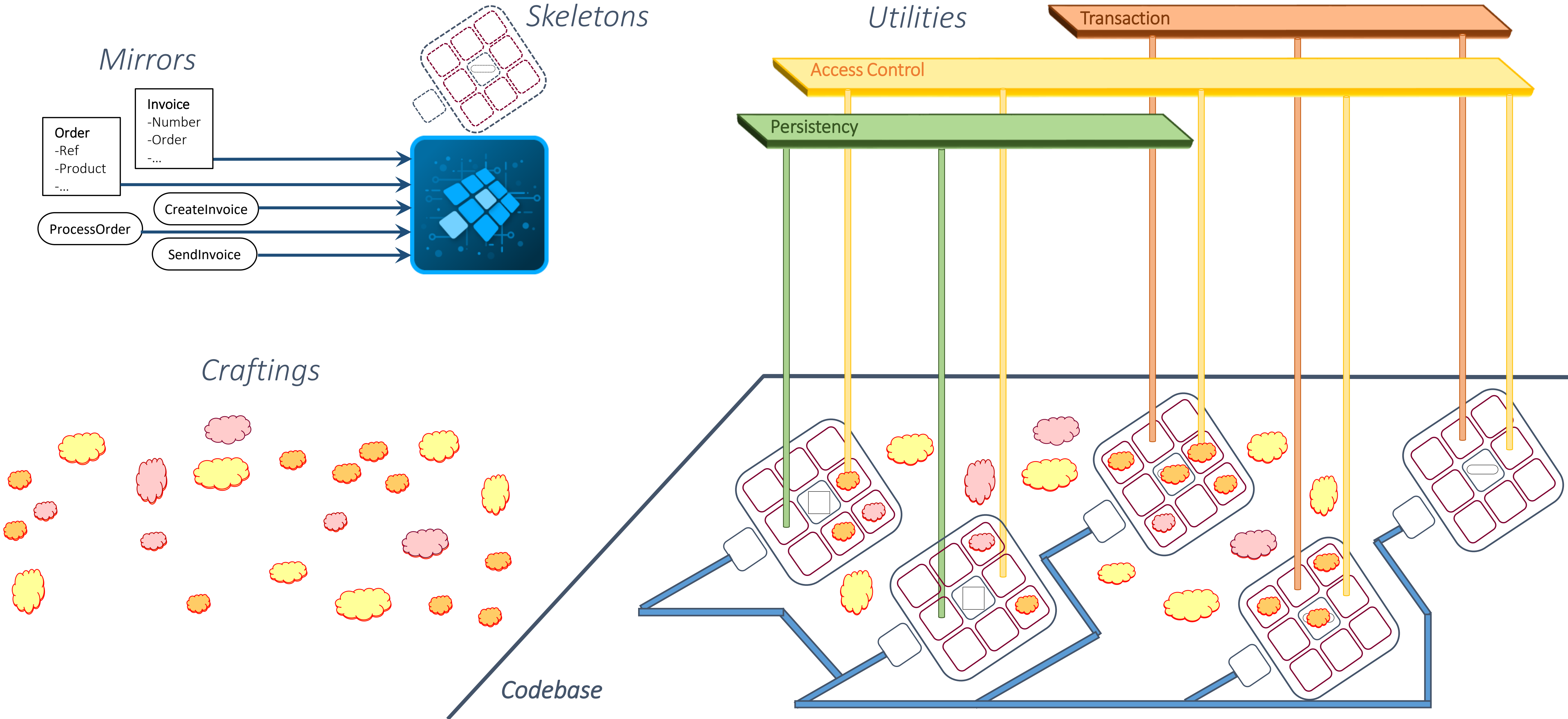
Change Dimension 2: The Utilities



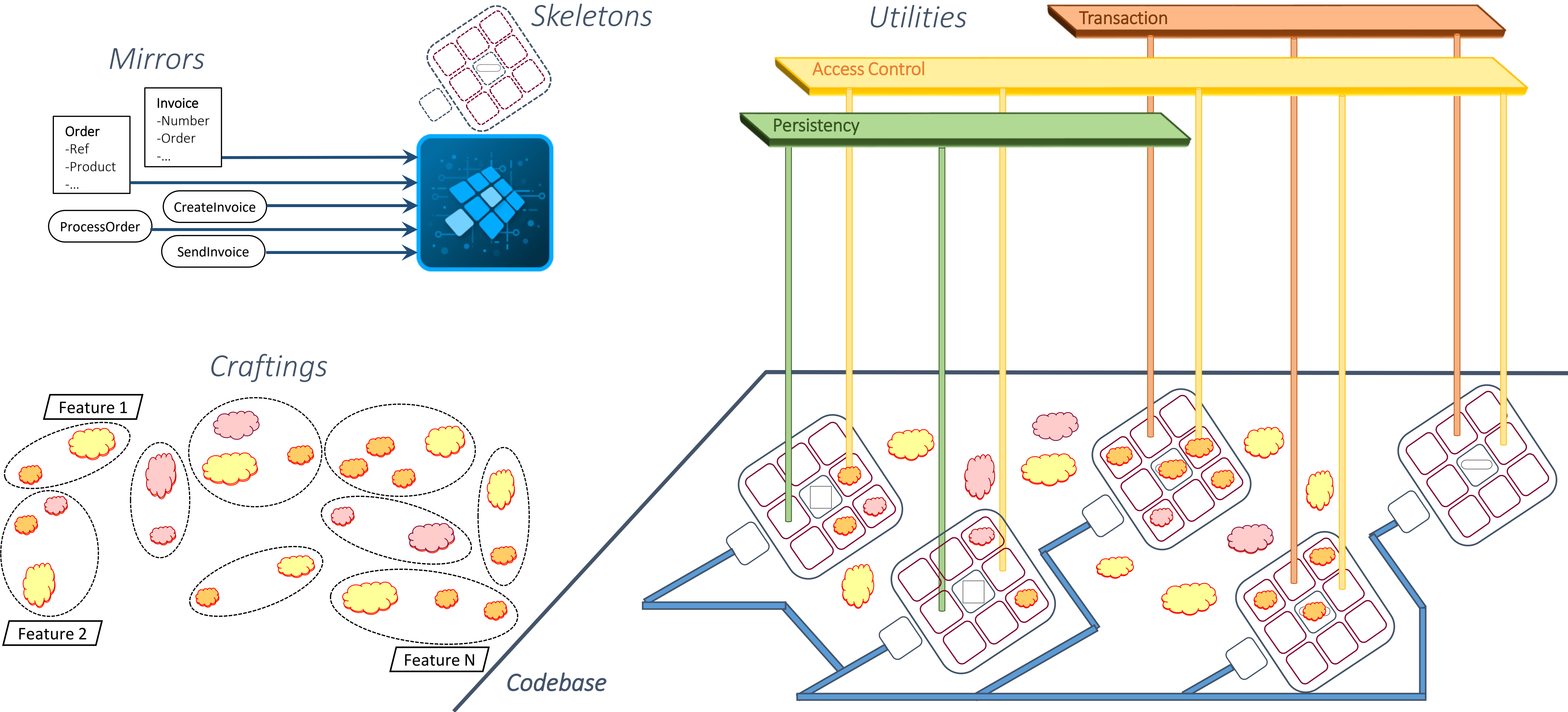
Change Dimension 3: The Skeletons



Change Dimension 4: The Craftings



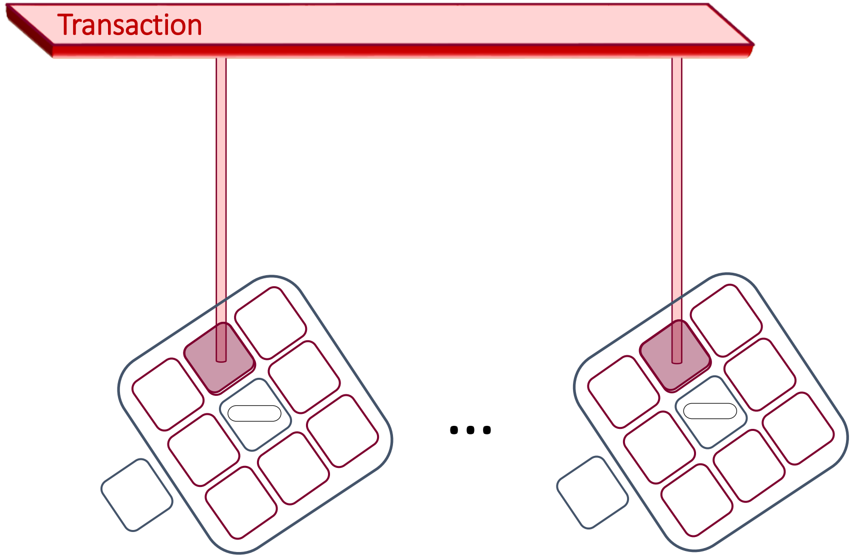
Change Dimension 4: The Craftings



Sustaining an Evolving Utility Landscape

Ex Ante 8 A technology implementation of a specific concern for one element, or a listed set of elements, can be changed in a stable way.

Change utility transactions

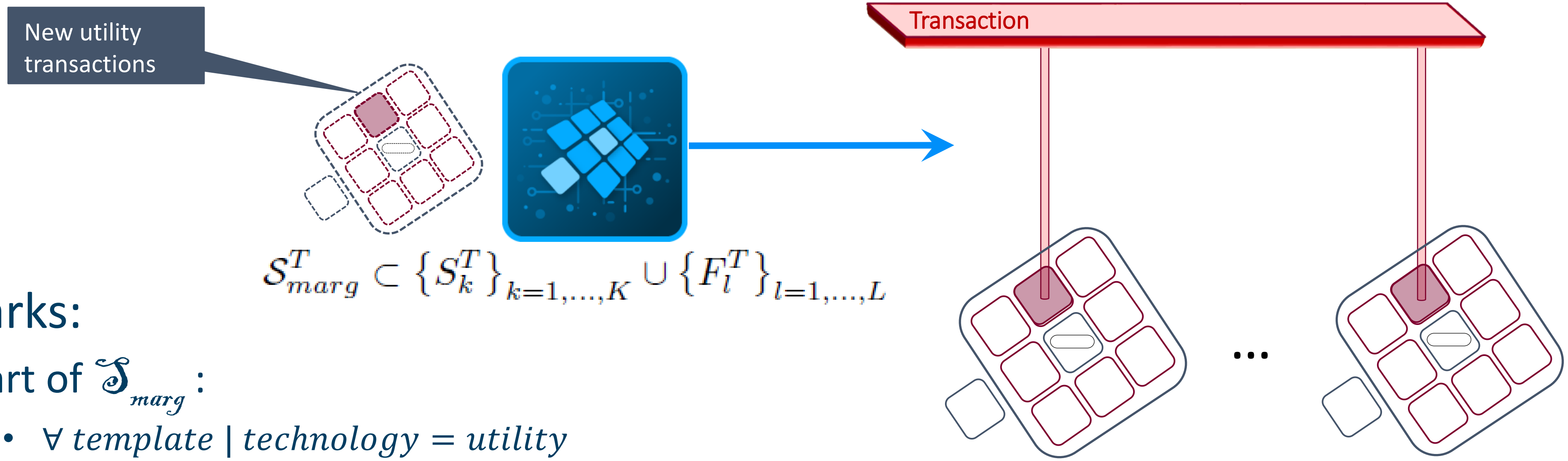


$$S_{marg} \subset \{S_{m,k}\}_{k=1,\dots,K} \cup \{F_{m,l}\}_{l=1,\dots,L}$$

- Remarks:
  - Part of $\mathfrak{S}_{marg}$:
    - $\forall\, artifact \mid technology = utility$
    - One for every listed element
  - Craftings : no direct utility calls

Sustaining an Evolving Utility Landscape

Ex Ante 9 – Expansion An additional technology implementation for a specific concern of a type of element, can be made available for all information systems in a stable way.

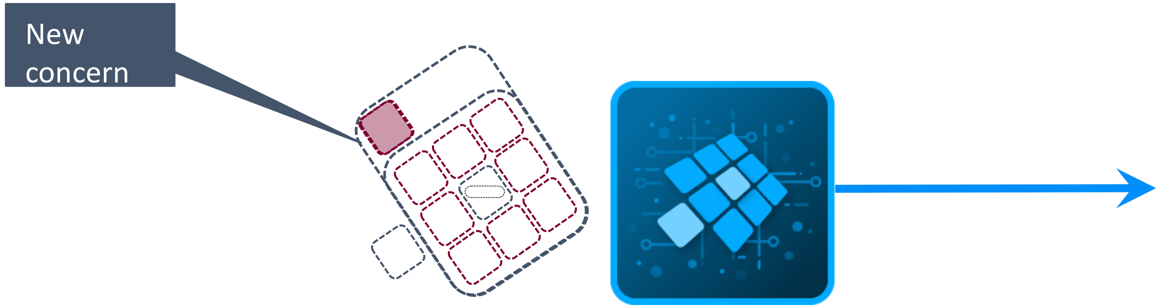


$$\mathcal{S}^{T}_{marg} \subset \{S^{T}_{k}\}_{k=1,\ldots,K} \cup \{F^{T}_{l}\}_{l=1,\ldots,L}$$

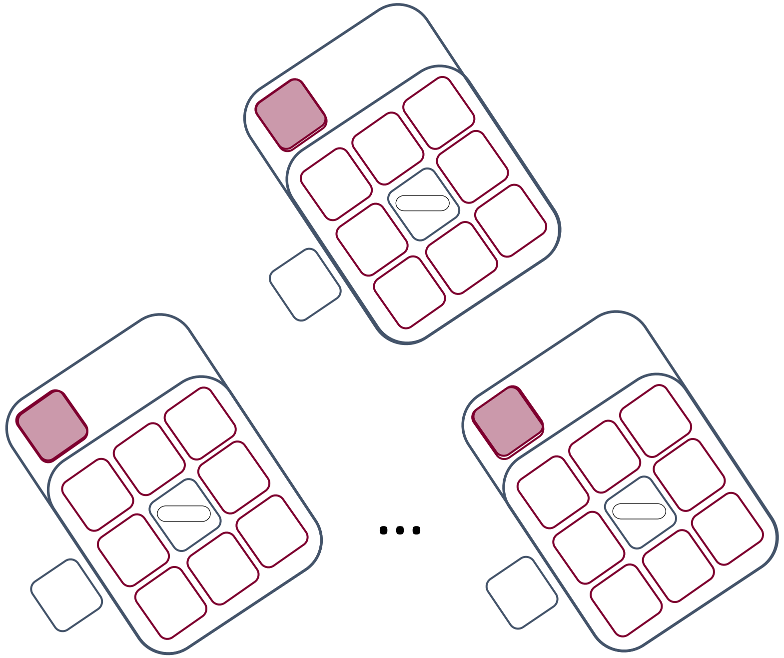
- Remarks:
  - Part of $\mathfrak{I}_{marg}$:
    - $\forall\ template \mid technology = utility$
  - Configuration :
    - Define setting or option
  - Craftings : no direct utility calls

Sustaining an Evolving Utility Landscape

Ex Ante 11 – Expansion An additional concern for an of element can be made available for all information systems in a stable way.



$$\mathcal{S}^T_{marg} \subset \{S^T_k\}_{k=1,\dots,K+\Delta_K} \cup \{F^T_l\}_{l=1,\dots,L+\Delta_L}$$



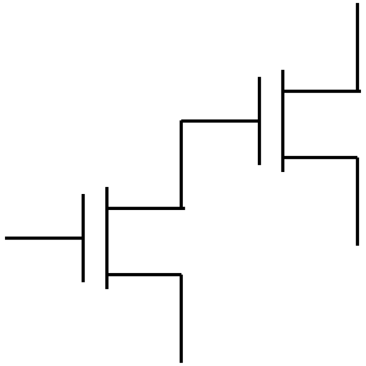
- Remarks:
  - Part of $\mathfrak{S}_{marg}$:
    - $\{\, templates \mid concern = concern \,\}$
  - Configuration :
    - Define setting or option

ESCAPING GROUNDHOG DAY

Overview

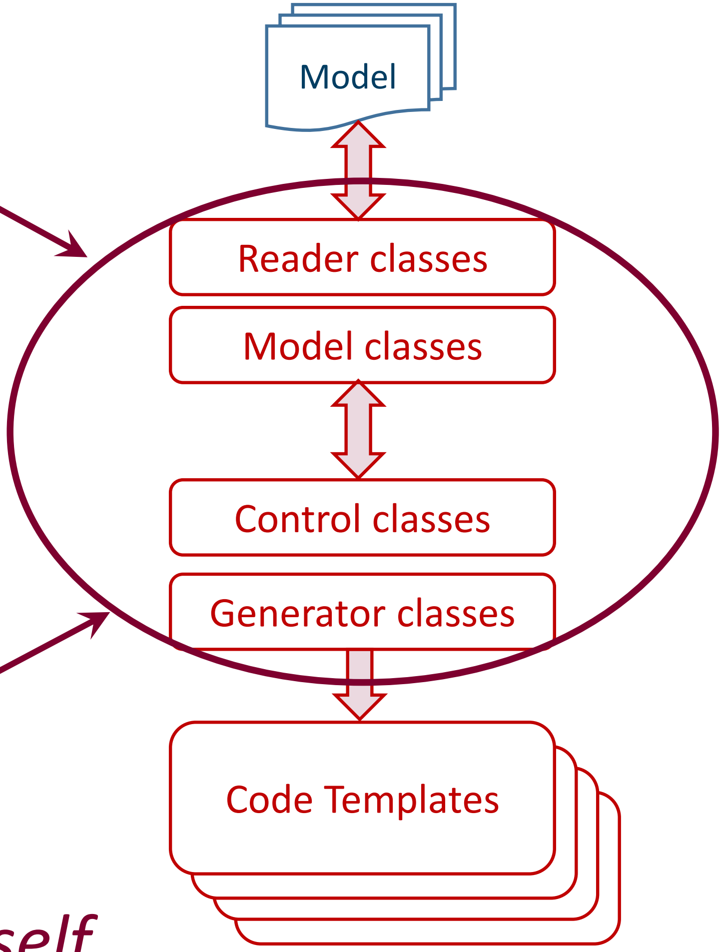
Meta-Circularity in Software Engineering

- Associated with terms like
  - *Homoiconicity*, coined in 1965, by Mooers & Deutsch (TRAC), uses concepts like *"code as data"* and *"program structure similar to its syntax*
  - *Meta-Circular Evaluator*, coined by John Reynolds in 1972, for an interpreter defining each feature of the defined language by using the corresponding feature of the defining language.
- Believed to *increase the abstraction level and therefore the productivity*
- Notion seems quite fundamental:
  - A transistor is switched by a transistor
  - A cell is produced by a cell

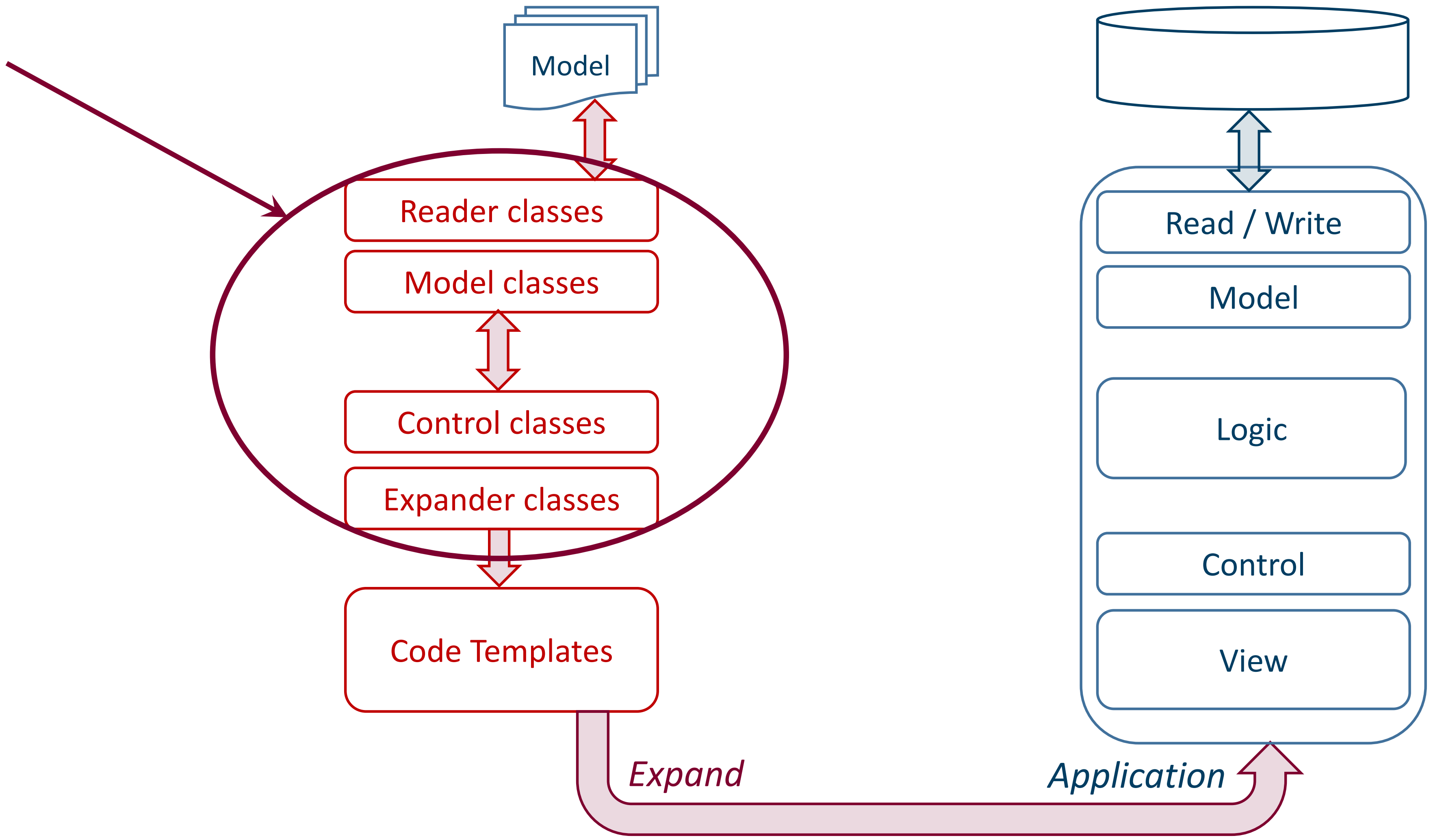


On Meta-Circularity in Metaprogramming

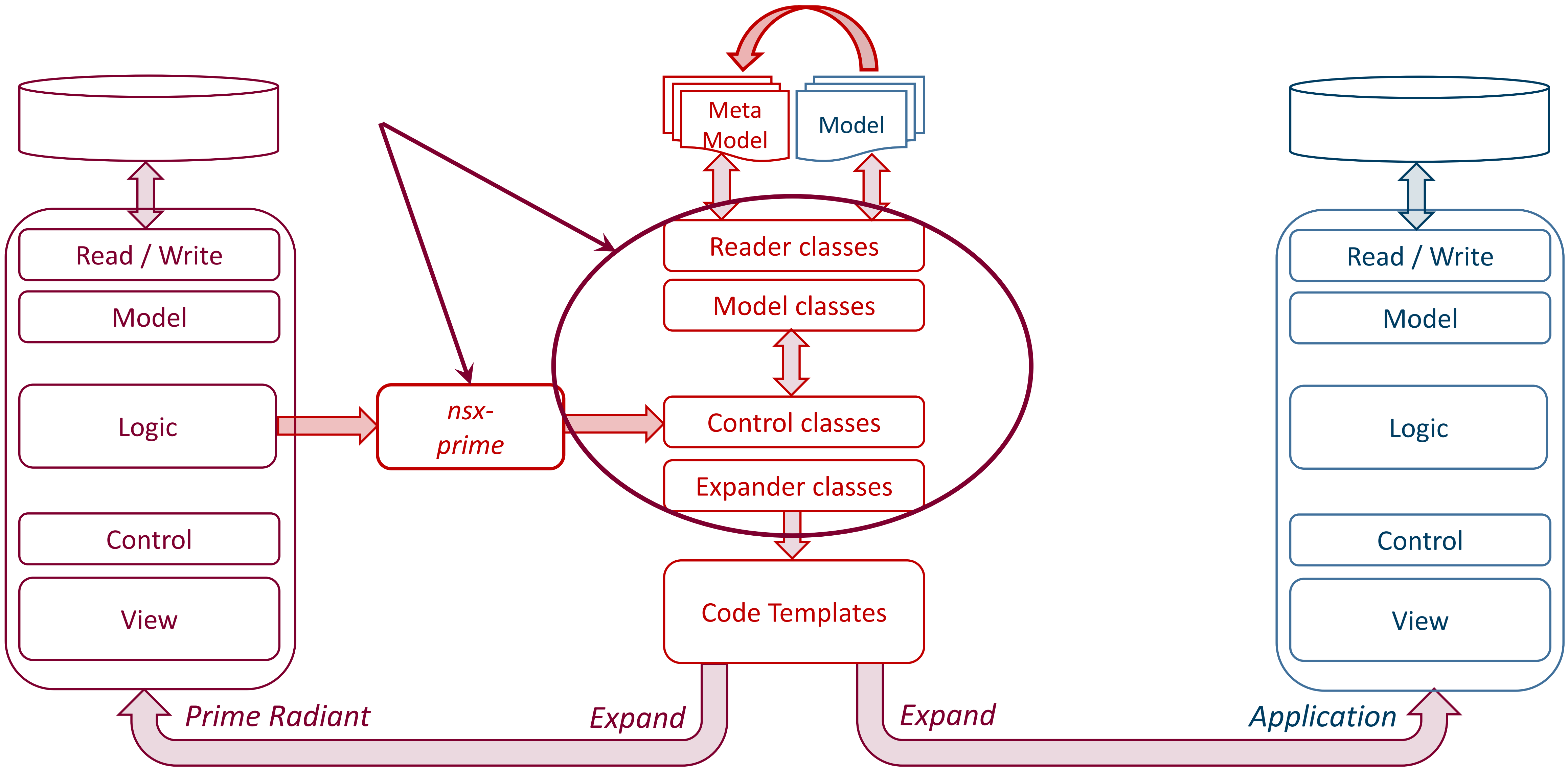
- You also have to maintain the meta-code
  - Consists of several modules
  - Is in general not trivial to write
- Will face growing number of implementations:
  - Different versions
  - Multiple variants
  - Various technology stacks
- Will have to adapt itself to:
  - Evolutions of its underlying technology
    - Which even may become obsolete
- *Meta-Circularity: meta-code that (re)generates itself*



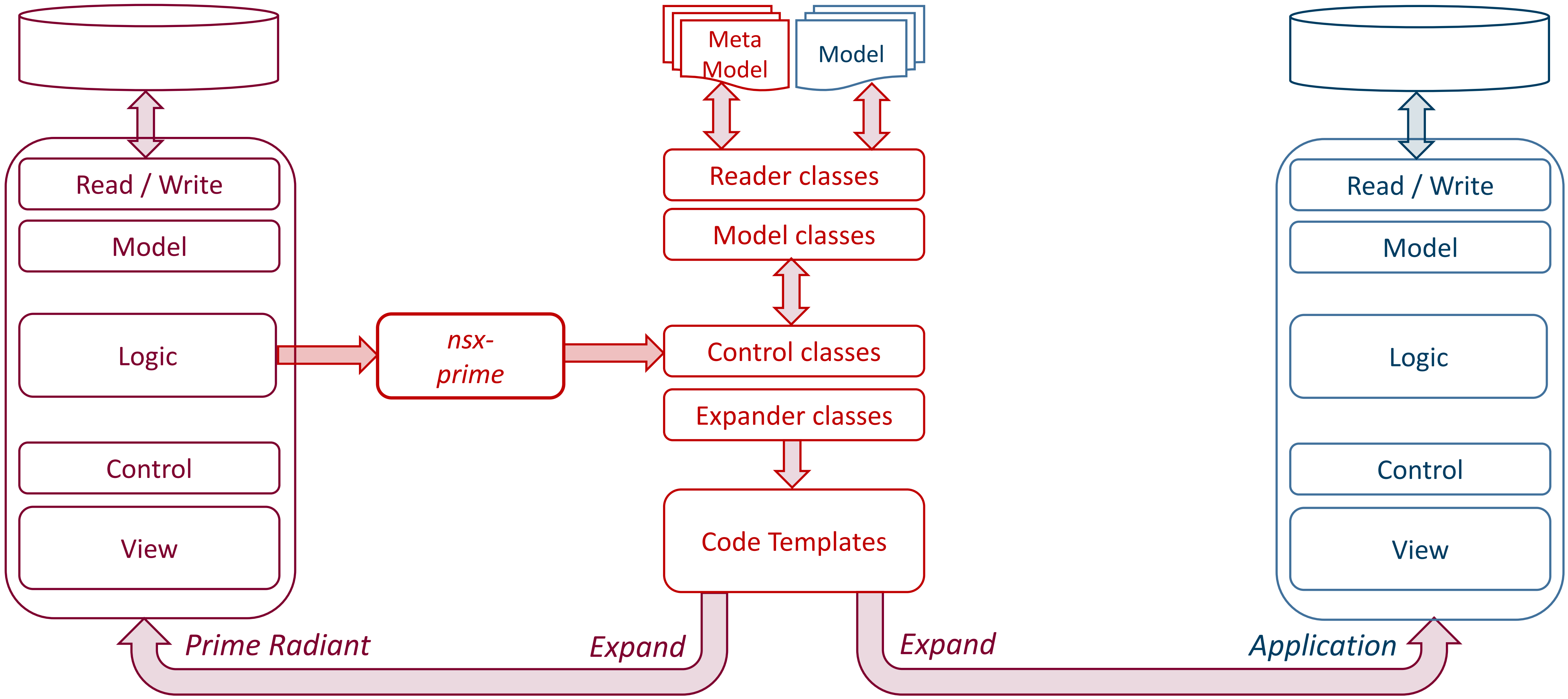
Establishing the Meta-Circle : Phase 1



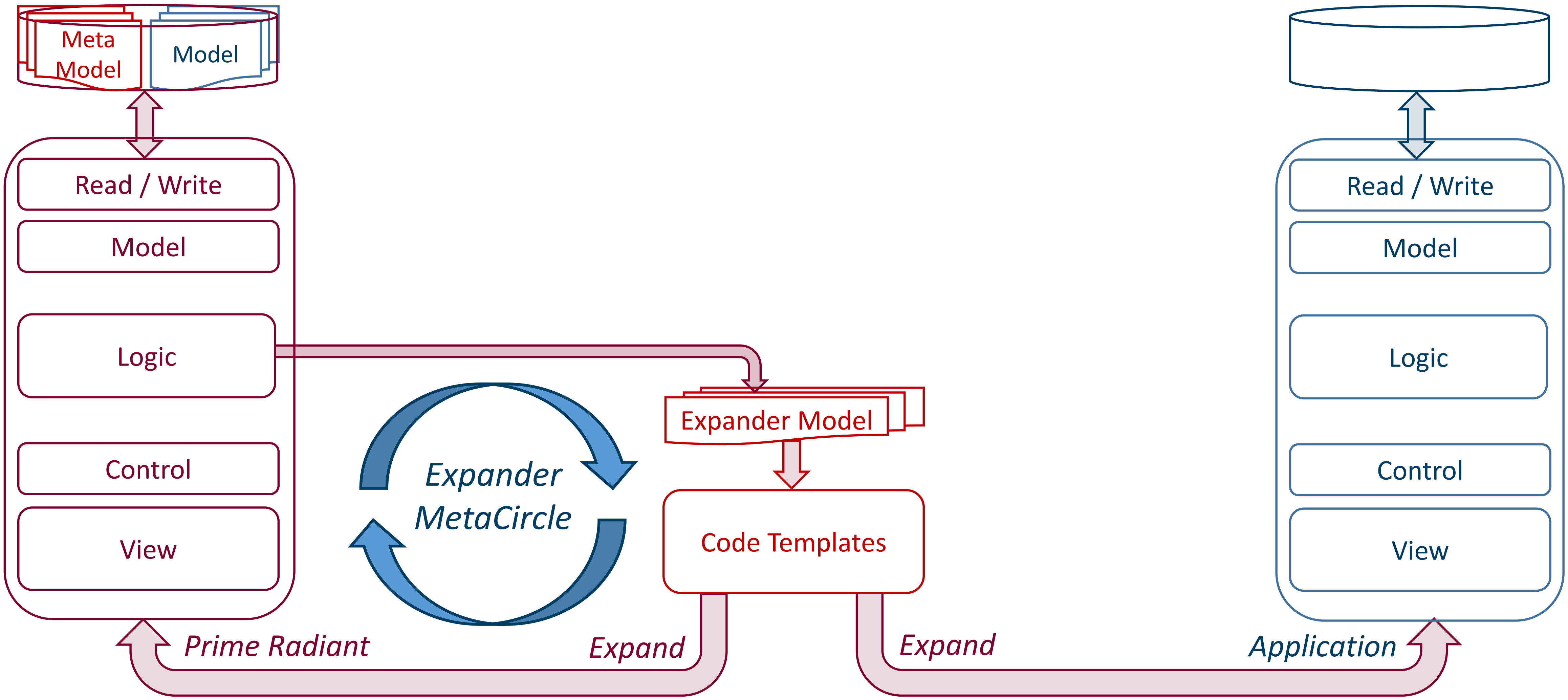
Establishing the Meta-Circle : Phase 2



Establishing the MetaCircle : Phase 3



Establishing the MetaCircle : Phase 3



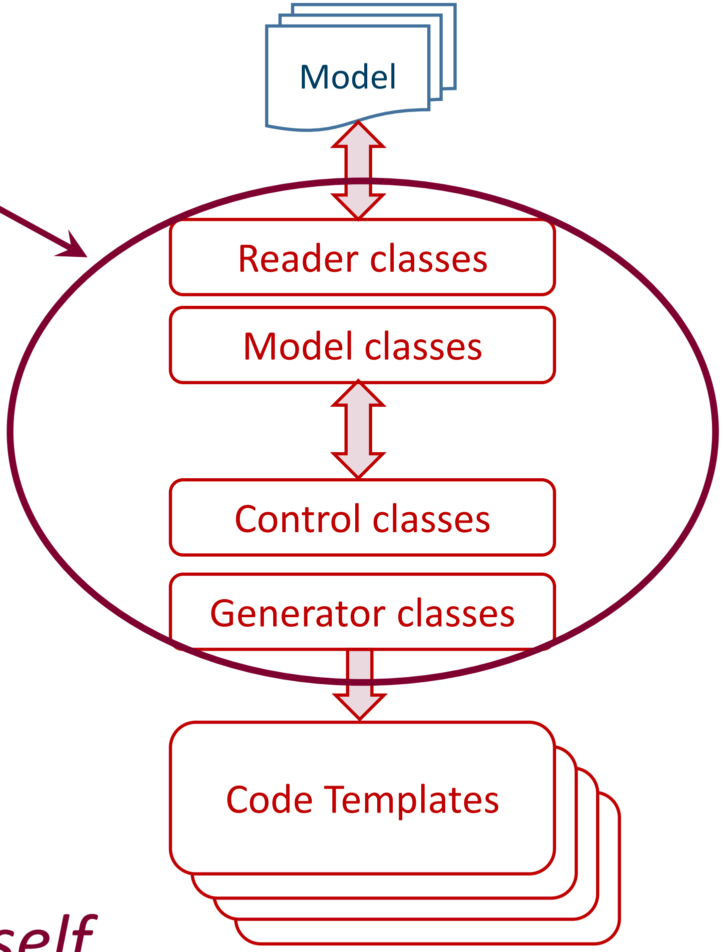
- Groundhog Day
- Foundations of Evolvable Software
- Toward Scalable Metaprogramming
  - Horizontal Integration
  - Realizing a Meta-ESB
- A Glimpse Beyond Software
- Conclusion

ESCAPING GROUNDHOG DAY

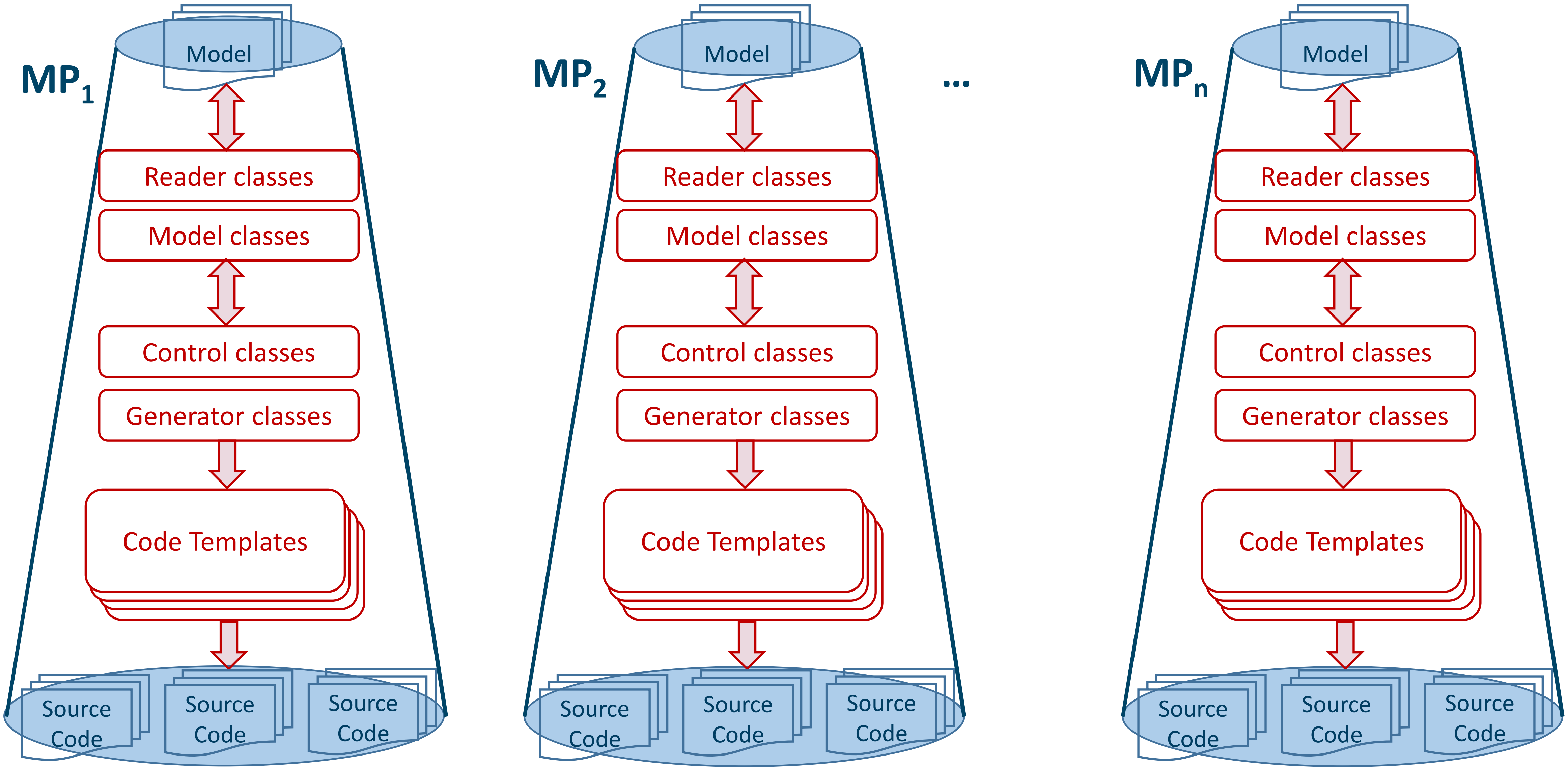
Overview

On Meta-Circularity in Meta-Programming

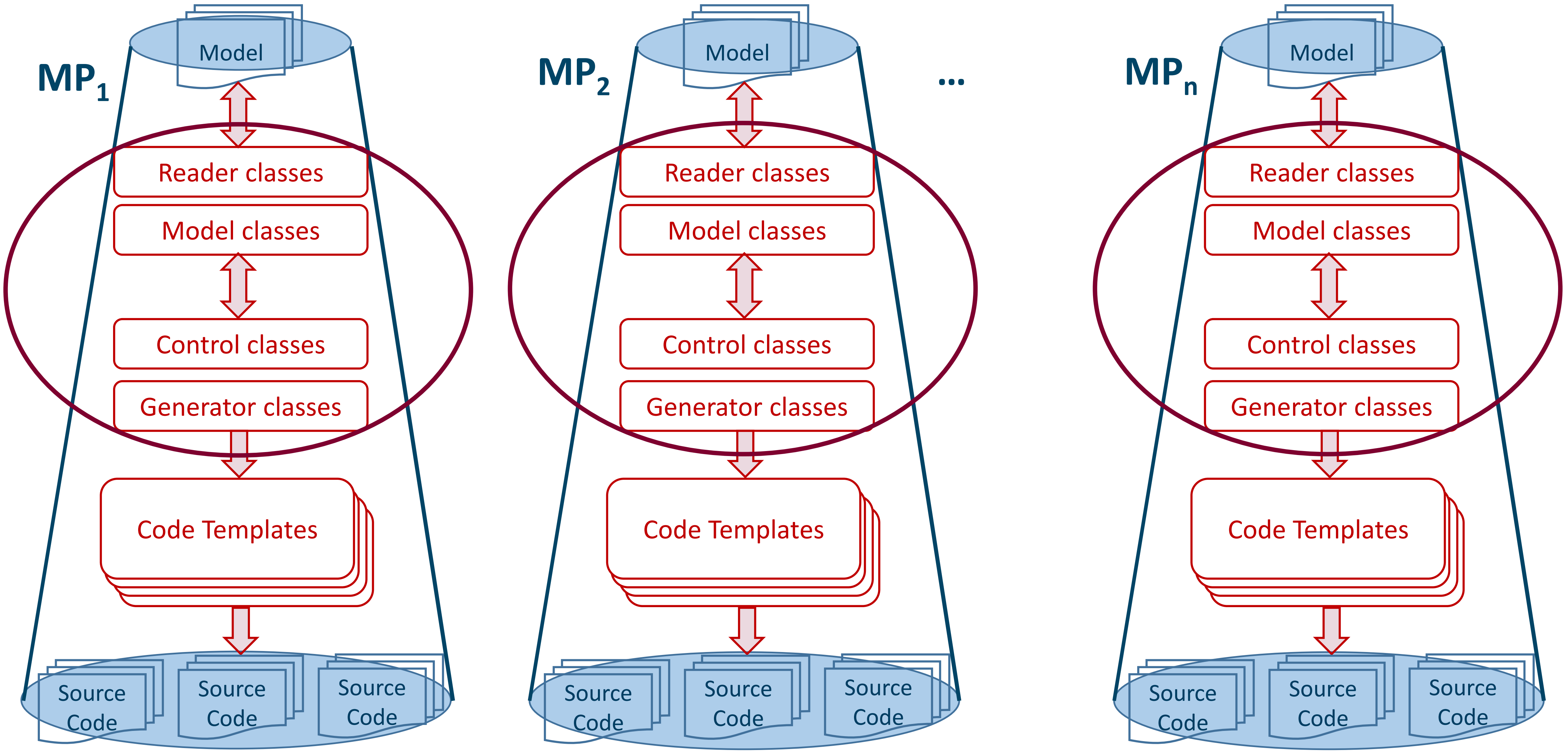
- You also have to maintain the meta-code
  - Consists of several modules
  - Is in general not trivial to write
- Will face growing number of implementations:
  - Different versions
  - Multiple variants
  - Various technology stacks
- Will have to adapt itself to:
  - Evolutions of its underlying technology
    - Which even may become obsolete
- *<u>Meta-Circularity</u>: meta-code that (re)generates itself*



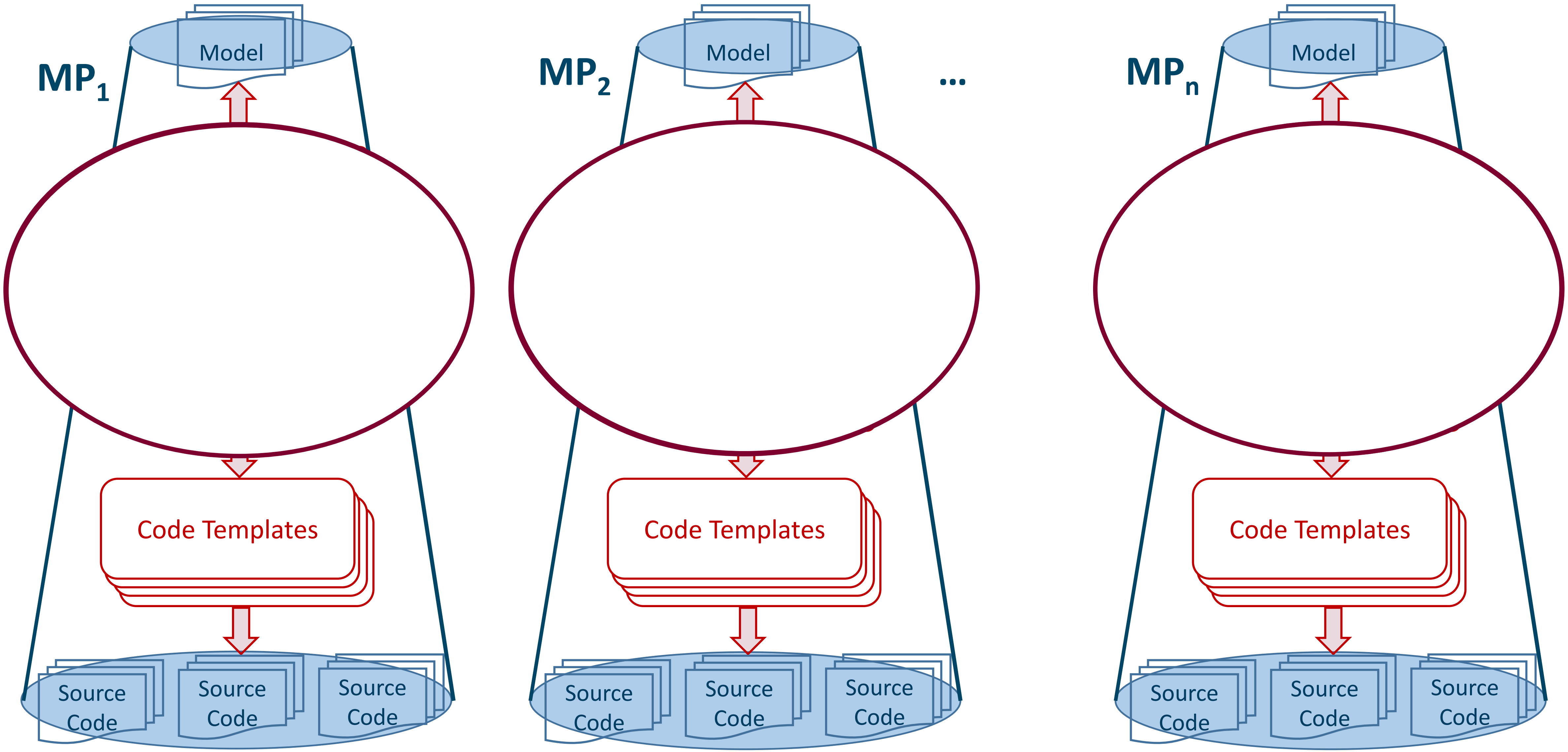
Vertical Integration or Metaprogramming Silos



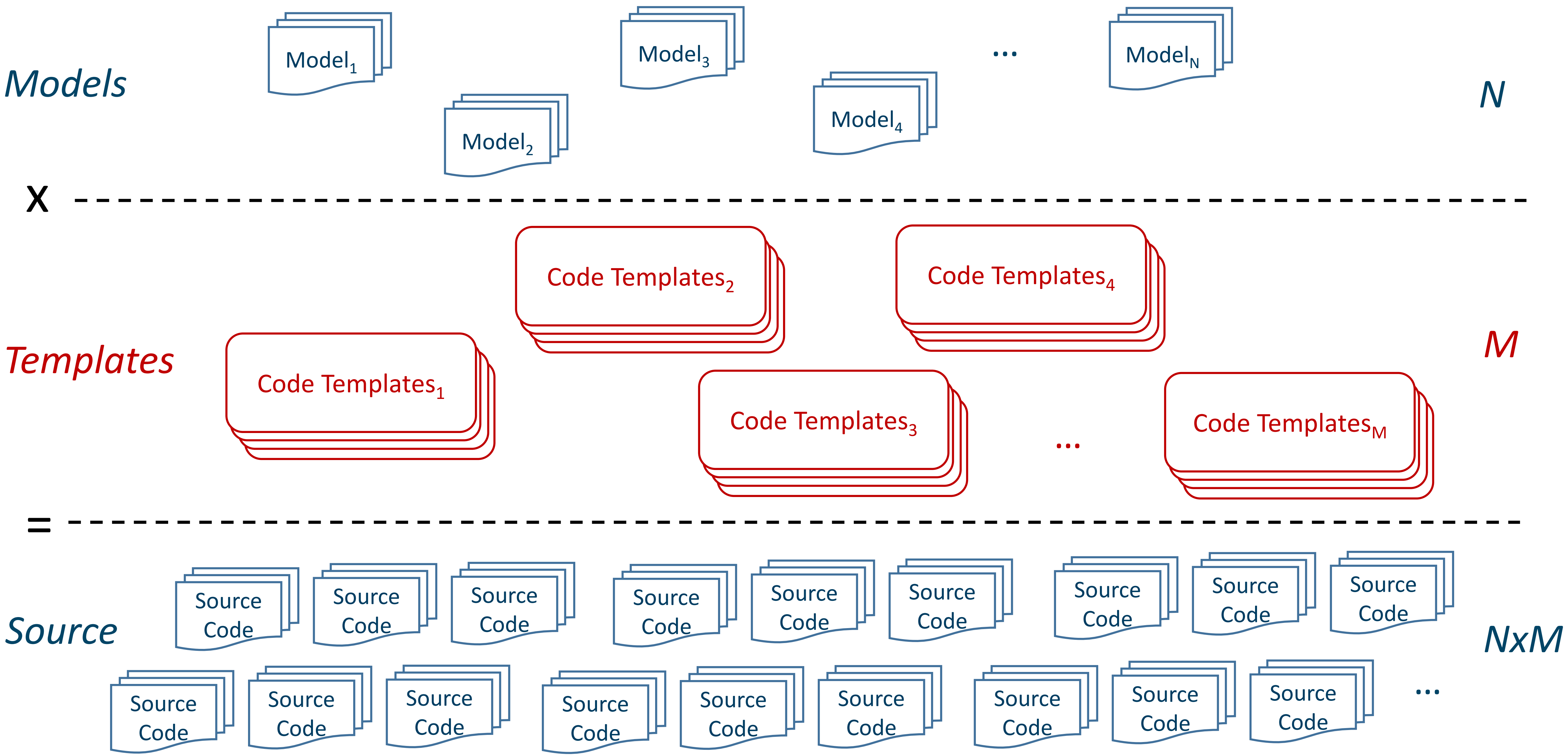
Vertical Integration or Metaprogramming Silos



To Horizontal Integration in Metaprogramming



On Horizontal Integration in Metaprogramming

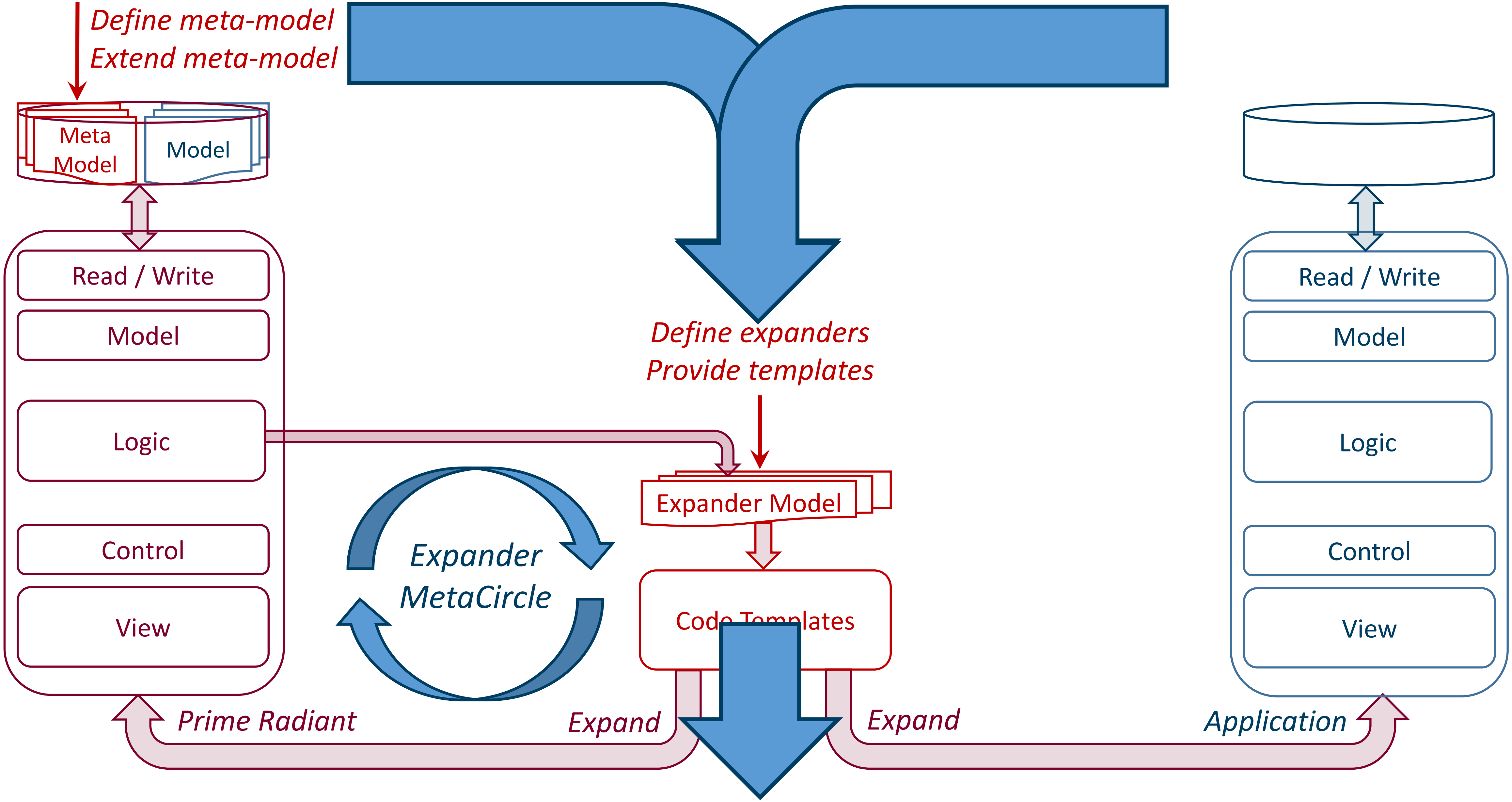


- Groundhog Day
- Foundations of Evolvable Software
- Toward Scalable Metaprogramming
  - Horizontal Integration
  - Realizing a Meta-ESB
- A Glimpse Beyond Software
- Conclusion

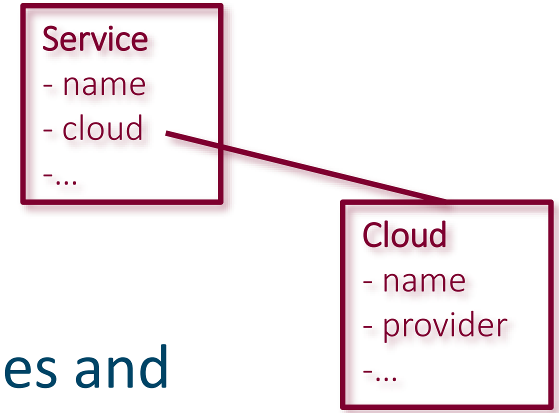
ESCAPING GROUNDHOG DAY

Overview

Metaprogramming Normalized Systems – Architecture

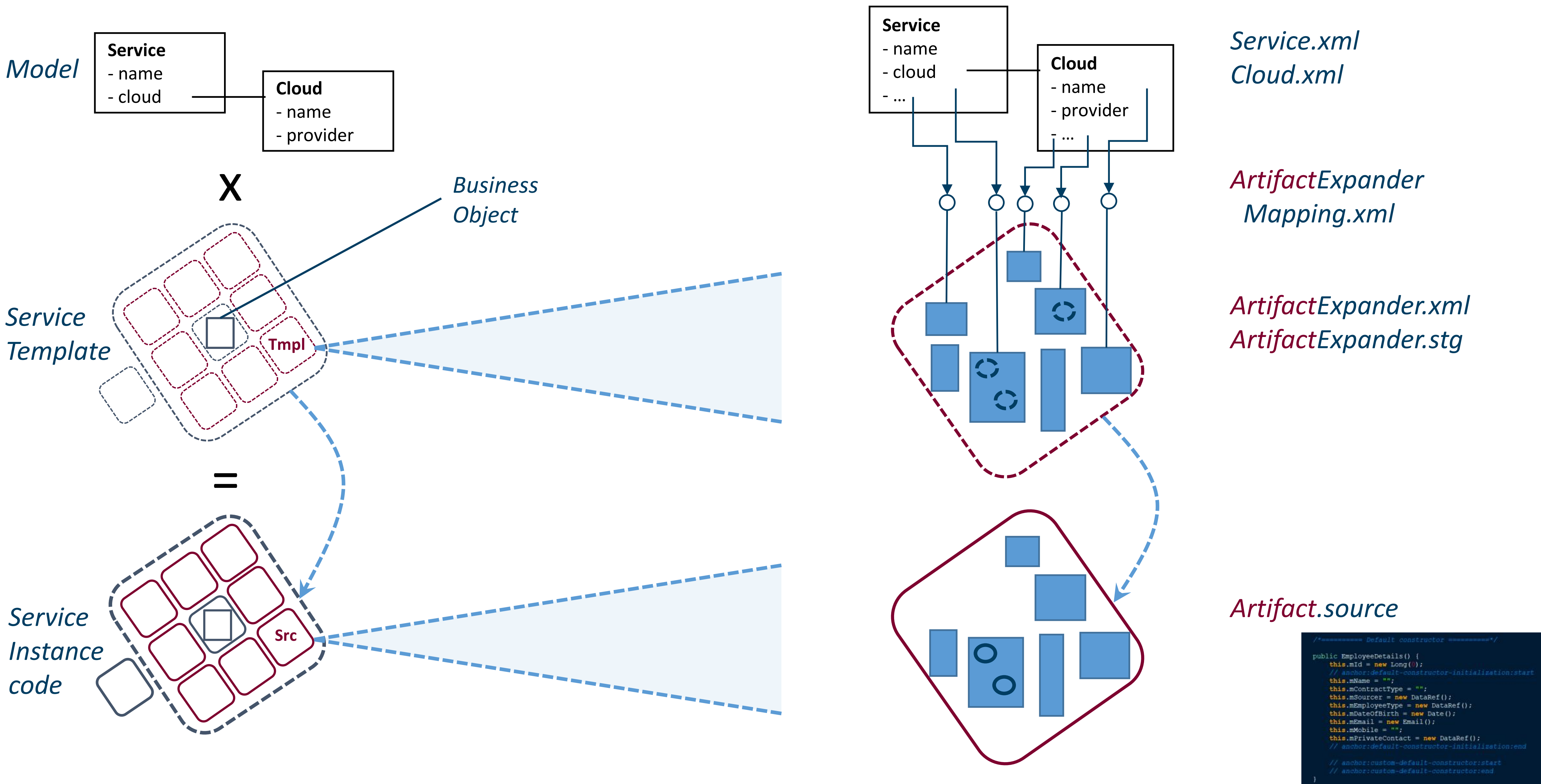


Integrating and Activating new Meta-Models



- Creation of a metaprogramming bus:
  - based on an horizontal integration architecture
  - using XML to exchange between models and templates
- Normalized Systems environment allows to:
  - define any Entity Relationship Diagram (ERD), including entities and relationships, e.g., *Service, Cloud*
  - generate the meta-circular stack for these entities, including:
    - XML readers and writers, e.g., *ServiceXmlReader*, *ServiceXmlWriter*
    - classes representing model instances, e.g., *ServiceDetails*, *ServiceComposite*
    - view and control classes for create and manipulate models in a user interface
  - make the models available to the templates through Object-Graph Navigation Language (OGNL) expressions
    - e.g., *service.name*, *service.cloud.name*, *service.cloud.provider*

Artifact = Expansion(Template, Model)

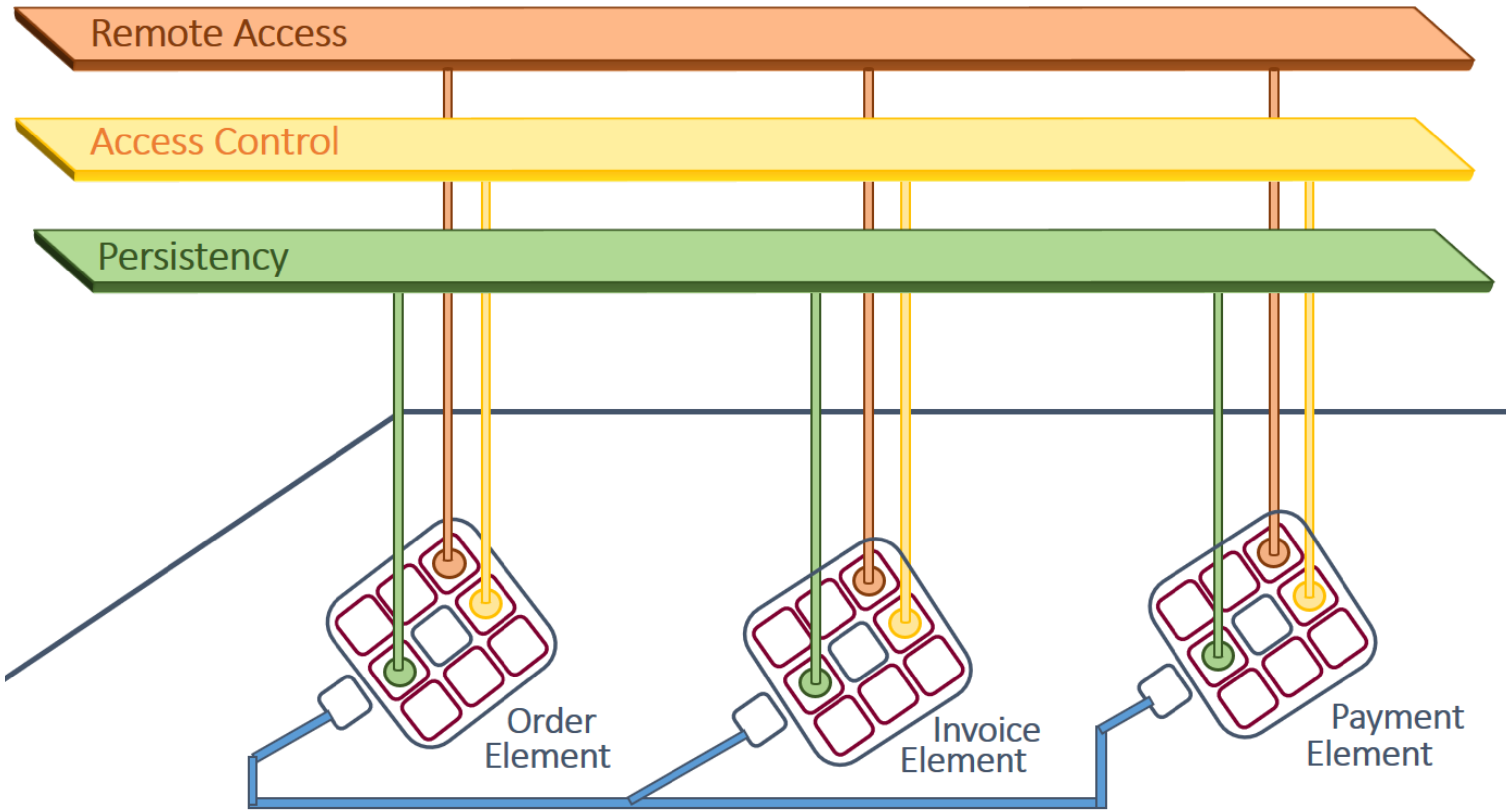


- Groundhog Day
- Foundations of Evolvable Software
- Toward Scalable Metaprogramming
- A Glimpse Beyond Software
  - Embedding Evolvable Utilities
- Conclusion

ESCAPING GROUNDHOG DAY

Overview

Guidelines on Cross-Cutting Concern

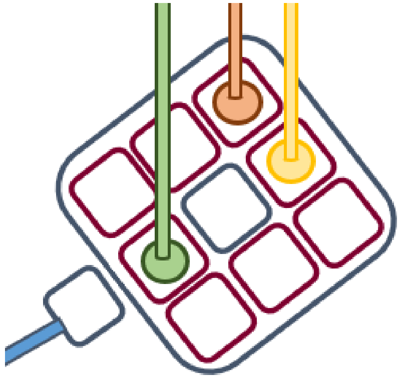


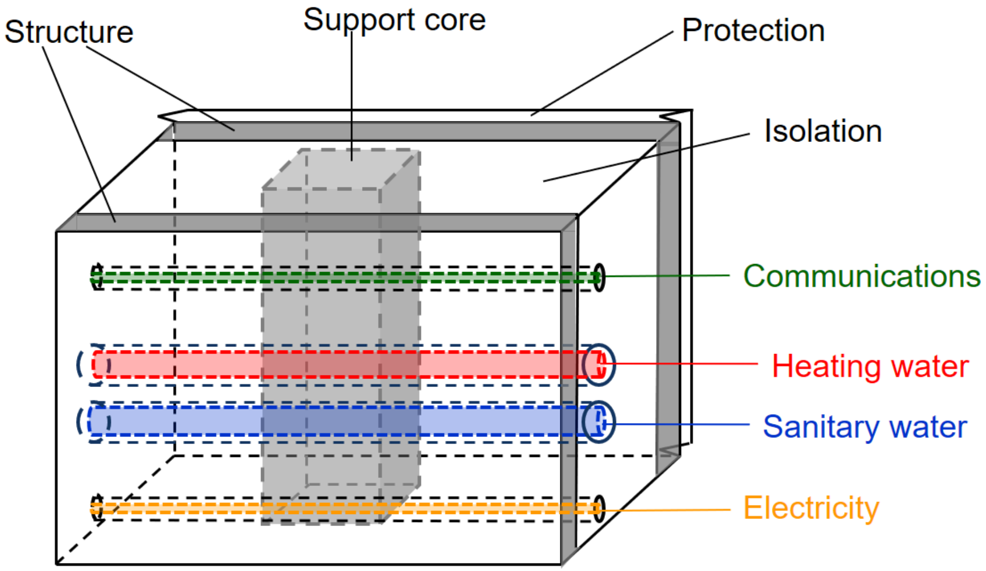
- *Encapsulation*
- *Interconnection*
- *Downpropagation*

A Basic Example: Heating

- Encapsulation
  - Fireplace → electric heater
- Interconnection
  - Fireplaces, electric heaters → central heating system
  - Central heating system → district and city heating
- Downpropagation
  - Central heating system → radiators in rooms
  - District and city heating → individual houses → rooms
  - *Can we propagate down to the individual “business objects” ?*

Some Construction Concept Elements





- Groundhog Day
- Foundations of Evolvable Software
- Toward Scalable Metaprogramming
- A Glimpse Beyond Software
- Conclusion

ESCAPING GROUNDHOG DAY

Overview

Conclusion

- We have indicated that the development of distributed landscapes of business functions may exhibit some tedious recurrent behavior
- Based on a thorough analysis, we have argued that evolvable software requires structured and preferably meta-circular metaprogramming
- We have explained how this meta-circular environment enables
  - the creation of dedicated meta-models to define parameters
  - to generate the infrastructure code to embed business objects in services
  - and to regenerate them in evolving cross-cutting concern landscapes
- While the metaprogramming environment has been successfully applied for years to (re)generate web-based information systems, the use of custom meta-models to generate service encapsulations is just starting

Some References

- Mannaert Herwig, McGroarty Chris, Gallant Scott, De Cock Koen, **Integrating Two Metaprogramming Environments : An Explorative Case Study :** ICSEA 2020 - ISSN 2308-4235 - IARIA, 2020, p. 166-172
- Mannaert Herwig, De Cock Koen, Uhnak Peter, **On the realization of meta-circular code generation : the case of the normalized systems expanders**, ICSEA 2019 - ISSN 2308-4235 - IARIA, 2019, p. 171-176
- De Bruyn Peter, Mannaert Herwig, Verelst Jan, Huysmans Philip, **Enabling normalized systems in practice : exploring a modeling approach**, Business & information systems engineering - ISSN 1867-0202 - 60:1(2018), p. 55-67.
- Mannaert Herwig, Verelst Jan, De Bruyn Peter, **Normalized systems theory : from foundations for evolvable software toward a general theory for evolvable design**, ISBN 978-90-77160-09-1 - Koppa, 2016, 507 p.
- Mannaert Herwig, Verelst Jan, Ven Kris, **Towards evolvable software architectures based on systems theoretic stability**, Software practice and experience - ISSN 0038-0644 - 42:1(2012), p. 89-116
- Mannaert Herwig, Verelst Jan, Ven Kris, **The transformation of requirements into software primitives : studying evolvability based on systems theoretic stability**, Science of computer programming - ISSN 0167-6423 - 76:12(2011), p. 1210-1222
- *<In progress>*, **On the Interconnection of Cross-Cutting Concerns within Hierarchical Architectures**, final review, IEEE Transactions on Engineering Management.
- **Normalized Systems Foundation Lectures** : https://www.youtube.com/channel/UCc8P1LREJogSlhwmAvdkq2A
- **Normalized Systems Prime Radiant Online:** https://foundation.stars-end.net **and** https://exchange.stars-end.net

QUESTIONS ?

herwig.mannaert@uantwerp.be